BACHELORARBEIT

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

Entwicklung von Frontend Single Page Webapplikationen mit dem modularen JavaScript-Framework AngularJS

Eingereicht von:	Patrick Müller
Geboren am:	10.09.1991 in Cloppenburg
Studiengang:	Angewandte Informatik
Matrikel:	BAIN11
Matrikel-Nr.:	18750

Erstprüfer:	Prof. Dr. rer. pol. Uwe Schröter
	Hochschule Merseburg (FH)
Zweitprüfer:	Christian Ernst, Dipl.-Inf.
	GISA GmbH (Halle)

Halle (Saale), den 10.09.2014

Kurzzusammenfassung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Entwicklung von Frontend Single Page Webanwendungen, basierend auf dem modularen JavaScript-Framework AngularJS. Dementsprechend werden die Technologien, Architekturmuster und die damit verbundene modulare Softwareentwicklung kritisch und praxisnah untersucht.

Abstract

This beachelor thesis deals with the deployment of frontend Single Page web applications based on the modular JavaScript framework AngularJS. Correspondingly the technologies, achritecture patterns and the associated modular development are critical and pratical under study.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, die vorliegende Arbeit selbstständig und ohne Zuhilfenahme unzulässiger Hilfsmittel angefertigt zu haben. Wörtliche oder dem Sinne nach übernommene Ausführungen sind gekennzeichnet. Diese Arbeit war bisher noch nicht Bestandteil einer Studien- oder Prüfungsleistung in gleicher oder ähnlicher Fassung.

Halle (Saale), den 10.09.2014 _____

Patrick Müller

Danksagung

Ich möchte mich zunächst bei allen bedanken, die mich während der Erstellung dieser Arbeit unterstütz haben.

Ein Dank gilt Herrn Prof. Dr. Uwe Schröter, für die gute Betreuung und die regelmäßigen Treffen. Daneben gilt mein Dank meinen Firmenbetreuer, Herrn Christian Ernst, der mich mit seiner konstruktiven Kritik und vielen Anregungen unterstützt hat.

Ganz besonders möchte ich mich bei meinen Eltern bedanken. Ohne sie wäre das Studium für mich unmöglich gewesen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Ajax	Asynchronous JavaScript and XML
API	Application Programming Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DI	Dependency Injection
DOM	Document Object Model
E2E	End-to-End
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
MVC	Model View Controller
MVVM	Model View ViewModel
OOP	Objektorientierte Programmierung
REST	Representational State Transfer
SPA	Single Page Webapplikation
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
XML	Extensible Markup Language

1 Einleitung

1.1 Motivation

Die Softwareindustrie ist seit Langem bestrebt, ihre komplexe Software auf einer breiten Palette von Systemen anzubieten. Wird Software nativ – also jeweils für ein (Betriebs-)System – entwickelt, so kann sie nur mit großem Aufwand auf vielen verschiedenen Systemen zur Verfügung gestellt werden. Besonders die wachsende Verbreitung von mobilen Endgeräten im Privatbereich und zunehmend im Businessbereich erschwert die Situation. Der Entwicklungsaufwand und die Kosten wachsen sehr stark mit zunehmender Softwarekomplexität.

Um dieses Problem einzudämmen, wechselte die Industrie von nativen zu browserbasierten Anwendungen. Flash, Silverlight und Java-Applets waren Lösungen für komplexe Frontend-Anwendungen im Web. Jedoch dämmten diese und ähnliche Technologien das Problem der Plattformunabhängigkeit nur begrenzt ein, da diese ohne Plug-ins für den Browser nicht funktionieren. [Developa]

Lange Zeit setze man daher auf serverbasierende Systeme, wie zum Beispiel JSP oder JSF. Hier wurde die Benutzeroberfläche zunächst komplett vom Server gerendert und dann fertig an den Client weitergereicht. [Müller 2010] Vorteil dieser Vorgehensweise ist, dass der Client dadurch entlastet wird. Es muss keine zusätzliche Software installiert werden. Der Haken dabei ist, dass der Server dadurch unnötig zusätzlich belastet wird. Hochdynamische Applikationen sind so nur schwer zu realisieren.

Die bessere Alternative bilden die modernen browserinternen (clientseitigen) Technologien wie HTML5, CSS und JavaScript. Diese Technologien laufen ohne Plug-in auf allen modernen Browsern, egal ob Desktop oder mobil. So ist es möglich, mit geringem Aufwand Software für eine Vielzahl von Systemen anzubieten. In Kombination mit serverseitigen Backendsystemen lassen sich sehr leistungsstarke Anwendungen entwickeln. [Developa]

Bei komplexen Anwendungen stoßen diese Technologien allein jedoch schnell an ihre Grenzen. Die Wartbarkeit und die Übersichtlichkeit leiden sehr schnell, die

Entwicklungskosten steigen und Fehler in der Software summieren sich. Um diesen Problemen entgegenzutreten, entstand in den letzten Jahren eine Vielzahl von Frameworks, die die Entwicklungsarbeit erleichtern.

Die GISA GmbH ist als IT-Dienstleister immer daran interessiert, ihr Know-how und damit ihr verbundenes Angebots-Portfolio zu erweitern. Kundenanforderungen ändern sich mit der Zeit und die Verwendung von aktuellen Technologien ermöglicht es, diesen gerecht zu werden. Daher ist die GISA bestrebt, neue Technologien und Vorgehensweisen zu verifizieren und gegebenenfalls einzusetzen.

Die Firma ist auch stetig bemüht, solche modernen Systeme kontinuierlich in ihre eigene Infrastruktur zu integrieren. Dadurch ermöglicht die GISA seinen Mitarbeitern ein optimales und modernes Arbeitsumfeld.

1.2 Ziel der Arbeit

Die vorliegende Arbeit soll aufzeigen, wie moderne Frontend-Webanwendungen mithilfe von AngularJS professionell entwickelt werden. Dabei sollen Vorteile gezeigt und mögliche Schwächen aufgedeckt werden. Die modulare Architektur des Systems wird betrachtet und dessen Schwierigkeiten verdeutlicht.

Essenziell ist Wahrung des Praxisbezugs. Dazu dient zur Veranschaulichung exemplarisch ein Prototyp, welcher als dynamische Webapplikation zur Reisezeitenerfassung von Mitarbeitern implementiert wird. Diese Applikation verwendet die in der Arbeit aufgeführten Technologien und Methoden. Somit wird diese Anwendung das theoretisch Betrachtete praktisch aufzeigen.

Um den Rahmen dieser Arbeit zu wahren, wird nur die clientseitige Implementierung samt Schnittstellendefinition betrachtet. Das serverseitige „Backend“ wird folglich nicht näher betrachtet.

1.3 Aufbau der Arbeit

Nach der Einleitung folgt im Kapitel 2 „Grundlagen“ eine Einführung in die theoretischen Grundlagen. Architekturen, Technologien und Vorgehensmodelle, die für das fundamentale Verständnis eine Rolle spielen, werden erläutert und kritisch betrachtet.

Im Kapitel 3 „AngularJS“ wird das AngularJS JavaScript-Framework vorgestellt. Es soll ein Überblick über die Arbeitsweise und die wichtigsten Elemente geschaffen werden. Das Framework wird dabei analysiert und seine Arbeitsweise veranschaulicht.

Im Folgenden Kapitel 4 „Prototyp“ wird die Reisebuchungs-Applikation für die GISA GmbH mit AngularJS entwickelt. Zunächst wird sie dafür im Unterkapitel 4.1 konzipiert. Infolgedessen wird die Anwendung im Kapitel 4.2 prototypisch umgesetzt. Die Reisezeitenerfassung dient als umfangreiches Beispiel einer Webapplikationen. Dieses Kapitel verdeutlicht den praktischen Nutzen des Frameworks und zeigt die Herausforderungen der Entwicklung praxisnah.

Abschließend wird in Kapitel 5 „Fazit“ das Ergebnis der Arbeit kurz zusammengefasst. Eine kritische Abschlussbetrachtung und ein kleiner Ausblick auf die mögliche Zukunft von Webanwendungen werden gegeben.

2 Grundlagen

2.1 Webanwendungen

Webanwendungen sind Softwaresysteme, die über das Inter- oder Intranet zur Verfügung gestellt werden. Die Anwendungen laufen im Browser des Benutzers und werden über einen Server (das Backend) in den Browser des Anwenders geladen. Bei modernen Webanwendungen teilen sich Client und Backend die Arbeit, das heißt, sie arbeiten im Verbund zusammen. [Horn]

Das bedeutet, der Client trägt einen nicht unerheblichen Teil zur Darstellung der Anwendung bei. Für die Anzeige und einen Teil der Verwaltung ist also der Browser verantwortlich. Der Client erhält die Anwendung vom Server/Backend die Rohdaten, mit denen er die Anwendung selbstständig ausführt, was dank moderner Web-Technologien[1] kein Problem darstellt. [Long Le]

Ein Großteil der Logik wird jedoch vom Backend getragen. Es ist für komplexere Berechnungen und für die Datenpersistenz verantwortlich. Der Komplexität des Backends sind theoretisch keine Grenzen gesetzt. Oft handelt es sich dabei um verteilte Systeme, bestehend aus z.B. Services, Datenbanken und Persistenz-systemen. [Long Le]

Client und Backend kommunizieren über das HTTP-Protokoll und den damit in Zusammenhang stehenden Architekturen, wie zum Beispiel REST[2]. [Horn] Das ermöglicht es, Anwendungen über das Web theoretisch weltweit zur Verfügung zu stellen.

Bei dieser Vorgehensweise ist zu beachten, dass die Anwendung eine bestehende Verbindung zum Server benötigt. Auch wenn das Frontend auf den Client, mithilfe von Speichermechanismen[3], dauerhaft (oder auf Zeit) gespeichert werden kann, ist die Anwendung offline nicht voll funktionsfähig.

[1] siehe Kapitel 2.2
[2] siehe Kapitel 2.6
[3] der „Cache“ oder der neue „localStorage“

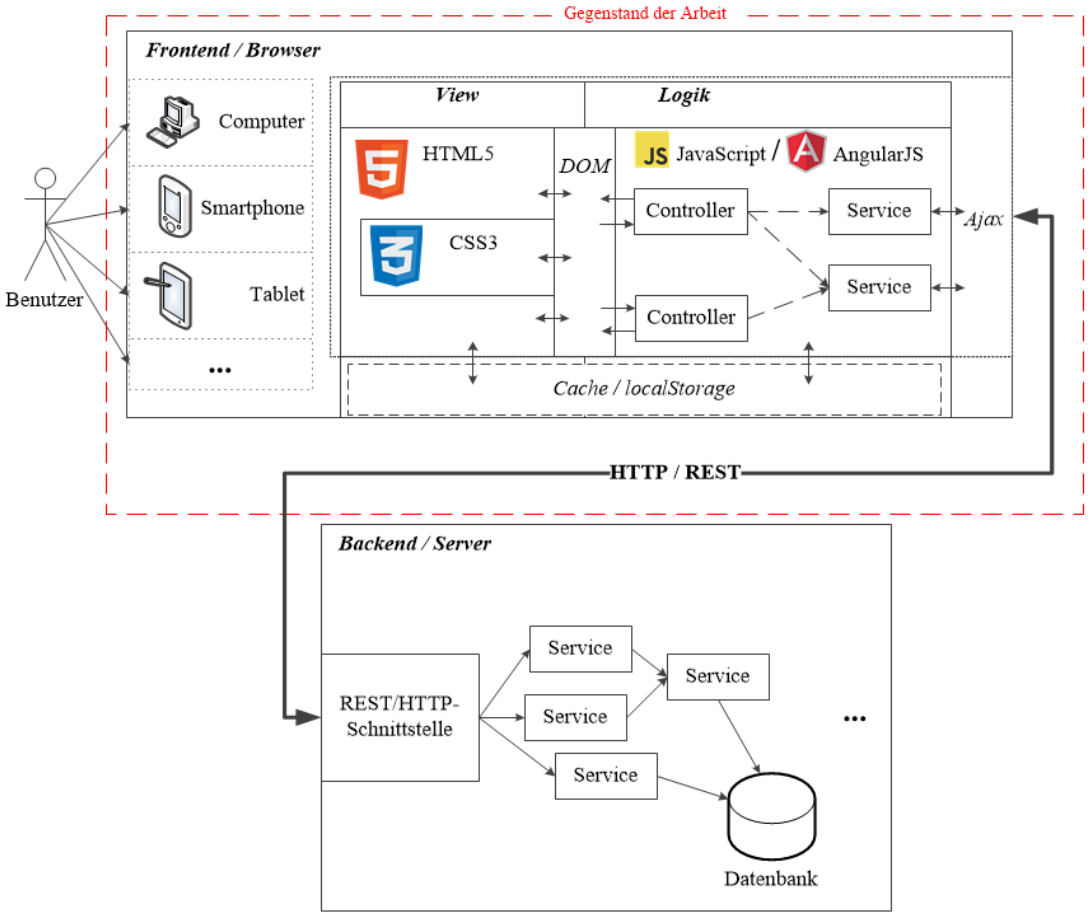

Abbildung 1: Illustration eines möglichen Aufbaus einer Webanwendung

Abbildung 1 veranschaulicht einen typischen Aufbau einer Webanwendung. Die Grafik zeigt das Zusammenspiel der wichtigen Web-Technologien[4]. Die Logik im Frontend kann innerhalb von realen Anwendungen selbstverständlich wesentlich komplexer sein. Die hier gezeigte Abbildung dient einzig und allein der Illustration des Prinzips.

Da das Backend in dieser Arbeit eine untergeordnete Rolle spielt, umfasst die Abbildung 1 auch nur eine vereinfachte Darstellung. So sind Backendsysteme in eingesetzten Anwendungen, wie bereits beschrieben, oft wesentlich komplexer. Auch der Ort, an dem sich das Backend befindet, spielt prinzipiell keine Rolle.

[4] siehe Kapitel 2.2

2.2 Web-Technologien

Grundlage der Implementierung von Webanwendungen bilden die Web-Technologien. Es handelt sich um eine Ansammlung von Techniken und „Sprachen“ die im Verbund arbeiten, um interaktive Benutzeroberflächen zu erstellen. Bei den wichtigsten Technologien handelt es sich um HTML5, JavaScript, die damit verbundene JSON und der DOM sowie Ajax.

HTML5 ist eine Auszeichnungssprache für (Web-)Dokumente, die aktuell parallel vom W3C[5] und der WHATWG[6] entwickelt wird. Die Auszeichnungssprache ist für die Strukturierung der Inhalte auf einer Webseite verantwortlich. Die neuste Version bietet neben neuen Elementen (sogenannten Tags) auch unter anderem APIs für Medien, „Geolocation“ und Offline-Anwendungen. [Wikipedia 1] Dies ermöglicht Entwicklern die Programmierung von modernen Webanwendungen für Desktop und mobile Clients.

Da es sich um eine Auszeichnungssprache handelt, ist HTML5 statisch. Um nun dynamische Anwendungen zu ermöglichen, ist die Zusammenarbeit mit anderen Technologien, wie JavaScript, nötig. Die neuen HTML5-APIs lassen sich mit HTML alleine auch nur begrenzt verwenden.

Bei **JavaScript** handelt es sich um eine, seit 1995 entwickelte, prototypenbasierte Skriptsprache, die innerhalb einer Webanwendung verwendet werden kann, um interaktive Anwendungen zu schreiben. Sie ermöglicht eine asynchrone Veränderung und Verarbeitung von HTML-Seiten über das Document Object Model (DOM). [Wikipedia 2] Solche Änderungen werden auch als DOM-Manipulationen bezeichnet.

Unter einer prototypenbasierenden Sprache versteht man eine Programmiersprache, die nach den grundlegenden Konzepten der Objektorientierten Programmierung (OOP) arbeitet. Jedoch existieren keine Klassen. Objekte werden hier zur Laufzeit durch Klonen vorhandener Objekte erzeugt. Diese Idee basiert auf dem Prototyp-Entwurfsmuster. [Wikipedia 3]

[5] http://www.w3.org/
[6] http://www.whatwg.org/

Mithilfe von JavaScript lassen sich komplexe, interaktive Anwendungen erstellen. Dabei übernimmt die Sprache die Steuerung der Anwendung und die Verarbeitung der Daten. Je dynamischer ein Programm wird, desto mehr Logik verweilt auf dem Client. Logischerweise muss folglich mehr JavaScript verwendet werden. Entsprechend ist es bei komplexen clientseitigen Anwendungen kaum mehr möglich auf diese Skriptsprache zu verzichten.

JSON ist ein kompaktes, objektorientiertes Datenaustauschformat zum Transfer von Objekten bzw. einer Liste von Objekten. Dabei werden Wertepaare, bestehend aus Schlüssel und Wert, verwendet. [Franke & Ippen 2012, S. 241]

Anwendung findet das Format unter anderem bei einem Datenaustausch zwischen Client und Server. Mithilfe von JSON ist es möglich, Objekte in kompakter und für den Entwickler in lesbarer Form zu übertragen. Eine schnelle und einfache Verwendung in JavaScript ist möglich. Objekte können auf diese Art auch innerhalb von JavaScript implementiert werden.

Ajax realisiert den asynchronen Datenaustausch von Client (Browser) und Server. Dadurch ist es denkbar, bei Datenabfragen nur bestimmte Teile einer Seite zu verändern, ohne die Seite komplett neu zu laden. Dies geschieht mithilfe von HTTP-Anfragen. [Theis 2014, S. 241]

Demnach ist es mithilfe von Ajax möglich, JSON-Dateien dynamisch zu laden und die Ansicht einer Anwendung partiell zu aktualisieren. Da nicht die gesamte Seite neu geladen werden muss, wird Traffic eingespart. Die Abarbeitung dieses Vorgangs ist schneller als ein komplettes Neuladen und spart entsprechend viel Netzwerk-Traffic. Das ist insbesondere bei der Verwendung von mobilen Endgeräten ein entscheidender Vorteil.

Das **Document Object Model (DOM)** ist eine Schnittstelle für den Zugriff auf XML- und HTML-Dokumente. Das Dokument wird dabei in einer Baumstruktur beschrieben, wobei die Knoten einzelne Elemente darstellen. [Theis 2014, S. 117]

Mit dem DOM ist es erst denkbar, HTML-Seiten mithilfe von Sprachen, wie JavaScript, zu verarbeiten. Dank der Baumstruktur ist es möglich, jedes Element performant zu bearbeiten. Operationen, wie Hinzufügen oder Löschen, stellen deswegen kein Problem dar.

2.3 Modulare Softwareentwicklung

Bei der modularen Softwareentwicklung geht es um die Aufteilung der Softwarelösung in einzelne Bestandteile, die als „Module“ bezeichnet werden. [Rainer Oechsle 2013, S.123f] Durch diese Splittung ist Software überschaubarer und leichter zu warten.

Module sind logisch oder funktionell abgeschlossene Softwareeinheiten zur Lösung einer bestimmten Aufgabe. Diese Einheit besteht aus Daten, Algorithmen und auch möglicherweise aus weiteren (Sub-)Modulen. [Isernhagen und Helmke 2004, S. 192]

Die detaillierte Implementierung eines Moduls ist dabei nicht von außen sichtbar (Blackbox). Das Modul kommuniziert nur über eine vordefinierte Schnittstelle nach außen. Dadurch hat eine Änderung innerhalb des Moduls keine Auswirkung auf dessen Verwendung. Diese Softwareeinheiten können theoretisch frei nach dem Baukastenprinzip zusammengesetzt werden. [Vishia]

Da die Kommunikation ausschließlich über Schnittstellen erfolgt, herrscht eine geringe Kopplung zwischen den einzelnen Modulen. Diese geringe Kopplung ist Voraussetzung für einige weitere wichtige Eigenschaften. Eine wichtige Rolle bei der Verwendung von Modulen spielt unter anderem die „Dependency Injection“[7] [Ulf Fildebrandt 2012, S. 100].

Ein Modul muss austauschbar sein. Das heißt, es muss durch ein anderes Modul (theoretisch) beliebig ausgewechselt werden können. Das Softwareprodukt als

[7] Siehe Kapitel 2.3.1

Ganzes soll dabei weiterhin voll funktionsfähig bleiben. Im Idealfall kann ein Austausch auch während des laufenden Betriebes erfolgen. Zu beachten ist nur, dass das neue Modul die gleichen Schnittstellen besitzen muss. [Rainer Oechsle 2013, S. 124]

Erweiterbarkeit und Reduzierbarkeit sind weitere Eigenschaften von Modulen bzw. modularen Systemen. Die Implementierung eines Moduls kann jederzeit erweitert werden. Folglich ist gewährleistet, dass die Software allzeit an neue Anforderungen angepasst werden kann. Auch nicht mehr benötigte Funktionalitäten können ausgebaut werden. Bei einer Reduktion ist zu beachten, dass es meist nur möglich ist, ganze Module zu entfernen. Modulinterne Funktionen können selten ohne Beeinträchtigung des Systems entfernt werden. [Rainer Oechsle 2013, S. 124]

Zu den interessantesten Eigenschaften gehört die Wiederverwendbarkeit. Alle Module sollen in anderen Systemen wiederverwendet werden können. [Barth 2012, S. 3] Diese Eigenschaft unterstreicht das Baukastenprinzip. Der Entwicklungsaufwand kann so erheblich gesenkt werden.

Durch die starke Aufteilung der (fachlichen oder technischen) Funktionalitäten ist es möglich, dass die Entwickler, unabhängig voneinander, Softwaremodule für ein System schreiben. Die Teamarbeit wird infolgedessen vereinfacht. Es ist möglich und empfehlenswert, die Module unabhängig voneinander zu testen. [Neumann & Murmann 2001]

Eine starke Ähnlichkeit zur **Objektorientierten Programmierung** ist anhand der Definition und der Eigenschaften erkennbar. Eine sauber entwickelte Klasse (inkl. Interface) erfüllt die wichtigsten Eigenschaften eines Moduls. Somit bringt die Verwendung der OOP zur modularen Softwareentwicklung sicherlich einige Vorteile, ist aber nicht zwingend nötig.

Trotz der sichtbaren Ähnlichkeit ist die modulare Softwareentwicklung bzw. Programmierung kein Synonym für die Objektorientierte. Mit Modulen sind meist übergeordnete Systeme gemeint, also z.B. eine größere Ansammlung von Objekten, die im Verbund arbeiten und nur über eine gemeinsame zentrale Schnittstelle nach

außen kommunizieren. Schließlich besitzt die OOP auch weitere Eigenschaften, wie z.B. die Vererbung, die in der modularen Entwicklung nicht definiert sind.

Aufgrund der fehlenden standardisierten Definitionen von **Komponenten** und Modulen werden beide Begriffe des Öfteren gleichbedeutend verwendet. Oft ist mit einer Komponente jedoch eine dem Modul übergeordnete Abstraktion gemeint. [Osten 2009, S8ff] Allerdings gibt es zugleich immer wieder Autoren, wie z.B. [Tarasiewcz & Böhm], die in einer Komponente einen untergeordneten Bestandteil (z.B. einen Service) eines Moduls sehen.

Dies stellt beim Architekturentwurf eine Herausforderung dar. Um eine genaue Abgrenzung zwischen Modul und Komponente zu schaffen, müsste eine einheitliche Definition vereinbart werden. Aktuell ist je nach Begriffsbestimmung der Unterschied zwischen beiden Systemen unterschiedlich groß. Eine detaillierte Definition ist von dem technologischen Umfeld abhängig.

2.3.1 Dependency Injection

Dependency Injection (DI) ist ein Entwurfsmuster zur Entkopplung einzelner Softwarebestandteile. Dabei werden die Abhängigkeiten nicht initialisiert, sondern sie werden von außen übergeben. Das ladende Modul muss dabei das verwendete Modul nicht im Detail kennen, der Injektor lädt die Implementierung automatisch in das Modul. Es entsteht eine Verbindung zur Laufzeit. [Goll und Dausmann 2013, S. 36f]

Dank DI ist es möglich, Module voneinander zu entkoppeln. Es ist ein elementarer Bestandteil der modularen Programmierung. So werden z.B. Services in Controller geladen. Auch das Laden von Modulen in andere Module ist möglich. [AngularJS-DI]

Folgendes Beispiel, soll das Prinzip verdeutlichen. Angenommen ein Controller möchte einen Service verwenden. Der Controller kennt jetzt lediglich die Schnittstelle des Services, den eigentlichen kennt er nicht. In diesem kleinen Beispiel (Abbildung 2, siehe Seite 11) verwendet der Controller „myCtrl“ die Service-Schnittstelle „IService“. Zur Laufzeit übergibt der Injektor dann den ihn bekannten

konkreten Service „myService“. Daraufhin kann der Controller den Service, wie in der Schnittstelle definiert, benutzten.

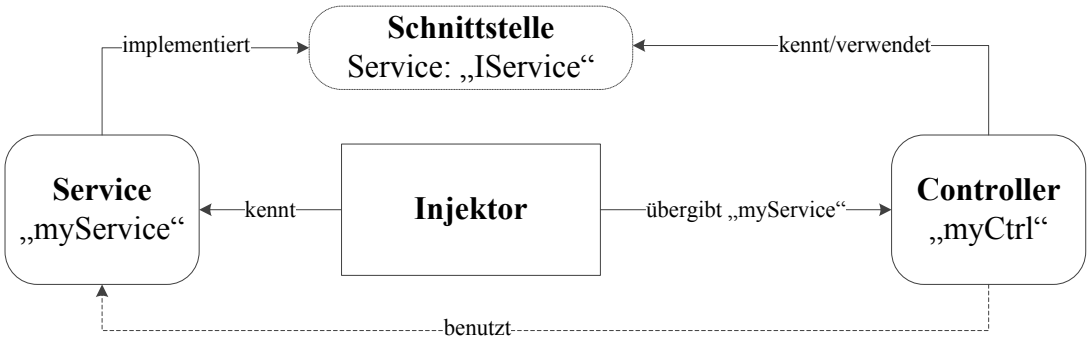

Abbildung 2: Beispiel Dependency Injection

Das dahinter stehende theoretische Grundprinzip ist die „**Dependency Inversion**“. Dabei handelt es sich um ein Entwurfsprinzip der objektorientierten Softwareentwicklung. Es geht darum, dass ein Modul ein untergeordnetes Modul nur über eine Abstraktion einbindet. Diese Abstraktion wird von dem übergeordneten Modul festgelegt. Dieses Modul ruft das Untergeordnete nicht mehr direkt auf, sondern nur dessen Abstraktion. [Goll und Dausmann 2013, S. 26f]

Dadurch ist gewährleistet, dass ein Modul nicht von seinen untergeordneten Modulen abhängig ist, sondern umgekehrt. Es entsteht eine geringere Kopplung und somit ist das untergeordnete Modul leichter austauschbar. [Goll und Dausmann 2013, S. 26f]

Problematisch ist hierbei die Tatsache, dass das höher liegende Modul die Schnittstelle vorgeben soll. In der Praxis ist dies häufig nicht der Fall. Wird zum Beispiel ein neues Modul entwickelt kann es passieren, dass ein bereits vorhandenes Modul verwendet werden soll. Dieses vorhandene Modul besitzt bereits eine Schnittstelle, die im Übergeordneten eingebaut werden muss. Die Abstraktion ist also bereits vom untergeordneten Modul vorgegeben und verstößt somit gegen das Prinzip.

2.3.2 Herausforderungen

Modularisierung bringt jedoch auch weitere Herausforderungen und Probleme mit sich. So ensteht ein Mehraufwand durch die benötigte Planung der Module. Es muss sichergestellt werden, dass es keine tiefen Abhängigkeiten der Module gibt. Ein hohes Wissen über den Aufbau der Anwendung ist nötig, wenn Änderungen vorgenommen werden sollen. [PDS Blog]

Eines der größten Probleme ist, dass sich in der Praxis verschiedene Aufgaben nicht unabhängig voneinander abarbeiten lassen. Module, die diese sogenannten „cross-cutting concerns“ beinhalten, bilden eine ungewollt starke Kopplung. Das sind (meist nichtfunktionale) Anforderungen, die sich nicht ohne Weiteres gebündelt abarbeiten lassen. Ein oft auftauchendes „cross cutting concerns“ ist das Aufzeichnen von Ereignissen (Logging). [Barth 2012, S. 7] Diese Problematik erschwert besonders die saubere Modularisierung.

Herausfordernd ist ebenso die Entscheidung, welche Art von Modulen man implementieren möchte. So können Module fachlicher Art sein, zum Beispiel ein Kundenverwaltungs-Modul oder ein Gewinnspiel-Modul. Alternativ ist es möglich ein rein technisches Modul zu implementieren, z.B. ein Controller-Modul oder ein Validation-Modul.

Ein rein technischer Modulaufbau, wie er z.B. im Angular-Seed[8] Anwendung findet, ist meist nur in kleineren Projekten zu finden. Bei größeren Projekten hat ein solches Vorgehen den Nachteil, dass sehr schnell die Eigenschaften von Modulen stark verletzt werden. Auch werden solche Projekte sehr schnell unübersichtlich und die Wartbarkeit sinkt.

Das wohl sinnvollste Vorgehen ist die Aufsplittung in fachliche Module. Die Eigenschaften der modularen Softwareentwicklung sind bei diesem Vorgehen in der Regel einfacher einhaltbar. Problematisch ist hierbei, dass technische Systeme, wie User Interfaces, nur schlecht in fachliche Module hinein passen.

[8] https://github.com/angular/angular-seed

Eine Variante, um dieses Problem abzuschwächen, wäre es beide Arten zu mischen. So kann die Anwendung im Kern aus fachlichen Modulen bestehen und zusätzlich technische Module zu Hilfe nehmen. So könnte z.B. ein Kontakt-Modul ein Validations-Modul zur Hilfe nehmen. Andere fachliche Module könnten dann auch einfach das Validations-Modul einbinden und verwenden, ohne das Kontakt-Modul zu kennen. Ersichtlich ist hierbei das Problem, dass die modulare Struktur komplex wird und durchaus ungewollte Abhängigkeiten entstehen.

Welche Probleme man eher in Kauf nimmt und welche nicht, ist reine Entwicklerphilosophie. Im Endeffekt ist absehbar, dass eine „Bilderbuch" Modularisierung kaum realisierbar ist.

2.4 Single Page Webapplikationen

Klassische Webanwendungen bzw. Webseiten bestehen aus „Seiten" und werden bei jeder Aktualisierung komplett neu geladen. Darunter zählen sämtliche Bibliotheken und Grafiken, sofern sie nicht in einem „Cache" geladen wurden. Dieses Verhalten ist sehr ungünstig, wenn z.B. nur ein kleiner Text aktualisiert wird. [Spindler 2014, S. 14f]

Single Page Webapplikationen (SPA) unterscheiden sich zu jener herkömmlichen Struktur in genau diesem Punkt. Sie laden nicht die komplette Anwendung neu, sondern nur die Teile, die sich verändert haben. Dies geschieht mithilfe einer Manipulation der HTML-Elemente über JavaScript. Bibliotheken, Grafiken und andere Elemente, die nicht betroffen sind, bleiben dabei vollkommen unangetastet. [Spindler 2014, S. 11f]

Da es keinen Seitenwechsel gibt, werden Ladezeiten erheblich verringert. Der Inhalt kann nach Bedarf schnell und flexibel geändert werden. SPAs geben dem Benutzer das Gefühl, das er eine „richtige" Anwendung vor sich hat. Eine exorbitante Stärke solcher Anwendungen ist die Darstellung von dynamischen Inhalten. [Spindler 2014, S. 12f]

Im Gegensatz zu klassischen Webseiten haben Single Page Anwendungen also keine „Seiten“, sondern werden nur über ihren Zustand gesteuert.[Osmani] In diesem Zusammenhang wird, z.B. um ein Login-Formular aufzurufen, nicht die Seite „login.html“ aufgerufen, sondern die Anwendung erhält den Zustand „zeige Login-Formular“. Dabei gibt es keine Beschränkungen, so kann der Zustand formal auch lauten „zeige Login-Formular und Gewinnspiel“, um weitere Inhalte anzuzeigen.

Wie bei der klassischen Variante spielen URLs eine wichtige Rolle. So kann der Zustand der Anwendung auch über die URL festgelegt werden. Dies stellt eine sinnvolle Ergänzung zum „unsichtbaren“ Zustand dar. Es ist also möglich, einen Hauptzustand über die URL zu definieren und weitere Anzeigedetails anwendungsintern (verborgen vor dem Anwender) festzulegen. [SPA Book]

Zu beachten ist, dass SPAs im Browser des Benutzers ausgeführt werden und eine Reihe von DOM-Manipulation beinhalten. Die große Anzahl an verschiedenen Browsern hat einen proportional hohen Testaufwand zur Folge. Besonders problematisch ist es, wenn ältere Browser unterstützt werden sollen. [SPA Book]

2.5 Architekturmuster

2.5.1 Model View Controller – Pattern

Das Model View Controller (MVC) – Pattern ist ein Architekturmuster für die Entwicklung interaktiver Systeme. Dabei wird das System in 3 Teile, sogenannte Komponenten, unterteilt. Das Muster ermöglicht Verarbeitung, Präsentation und die Steuerung der Eingaben (bzw. Ausgaben) getrennt voneinander zu verwalten. [Goll und Dausmann 2013, S. 377]

Das **Model** umfasst die Geschäftslogik und -daten. Die Komponente ist unabhängig von der View und dem Controller. [Goll und Dausmann 2013, S. 380f] Dies ermöglicht es, das Model unabhängig zu entwickeln. Eine Wiederverwendung in

anderen Anwendungen ist möglich, ohne dass Änderungen an der Logik oder der Daten vorgenommen werden müssen.

Die **View** ist die Darstellungsschicht der Benutzerschnittstelle. Sie beinhaltet die für die Darstellung der Daten benötigten Elemente. Sobald Daten geändert werden, kann sich die View im sogenannten Pull-Betrieb aktualisieren. [Goll und Dausmann 2013, S. 381f] Somit ist gewährleistet, dass die View immer die aktuellen Daten beinhaltet. Um dies zu ermöglichen, muss die View das Model kennen. Es existiert also, aus Sicht der View, eine Abhängigkeit zum Model. Welche Daten in welcher Form dargestellt werden, kann beliebig variiert werden.

Die letzte Komponente ist der **Controller**. Er steuert die Ereignisse und entscheidet, was mit Benutzereingaben geschieht. Der Controller ist für die Veränderungsaufforderung der Daten im Model verantwortlich. Zu jeder vorhandenen View muss ein entsprechender Controller existieren. Diese Steuereinheit kann die View auch über ereignisabhängige Änderungen informieren. [Goll und Dausmann 2013, S. 380f] Daraus schließt sich eine starke wechselseitige Abhängigkeit zwischen Controller und View, da der Controller die View genau kennen muss und umgekehrt.

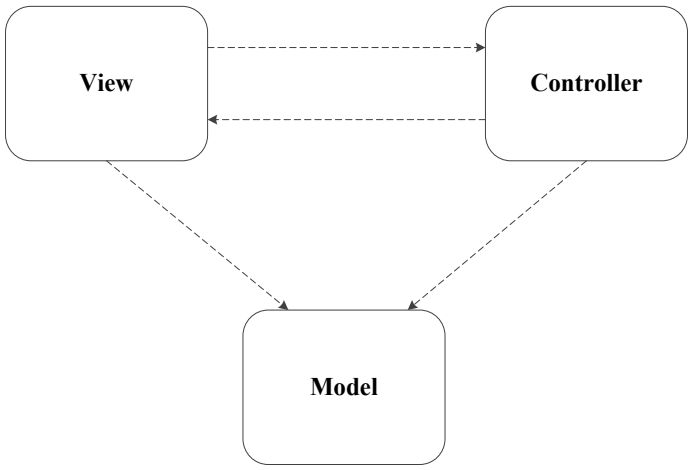

Abbildung 3: Abhängigkeiten der MVC-Architektur (vgl. [Goll und Dausmann 2013, S. 380])

Der Benutzer sieht die View und löst Ereignisse (Events) im Controller aus. Dieser steuert die darauf folgenden Abläufe. Das heißt, er kann z.B. die Daten eines Models ändern oder eine neue View öffnen. Das Model kann, ohne die View genau zu kennen, die entsprechenden Views über Änderungen informieren. Diese können darauf, die neuen Daten anfordern. [Goll und Dausmann 2013, S. 379]

2.5.2 Model View ViewModel – Pattern

Das Model View ViewModel – Pattern ist ein auf das MVC – Pattern basierendes Architekturmuster. Dieses Pattern zielt darauf ab, eine noch stärkere Trennung zwischen User Interface und Logik aufzubauen. Eine besondere Rolle spielt dabei das „Data Binding“. [Wikipedia 4]

Innerhalb dieses Musters nimmt das **Model** die exakt gleiche Rolle ein. Es ist theoretisch sogar ohne Weiteres möglich, ein Model aus einer im MVC realisierten Anwendung in einer MVVM Umgebung zu verwenden.

Die Darstellungsschicht, genannt **View**, beinhaltet alle Elemente der grafischen Benutzeroberfläche. Anders ist jetzt nur, dass die View ihr Model nicht mehr kennen muss. Eine Verbindung der View mit dem Model erfolgt über das ViewModel in der Form von Eigenschaften (Properties). [Wikipedia 5]

Das **ViewModel** ist das abstrakte Model der View. Es ist die verbindende Schicht zwischen Model und View. Sie sorgt dafür, dass die benötigten Daten in korrekter Form in der View angezeigt werden. Diese Schicht ist auch dafür verantwortlich neue Inhalte automatisch aus der View in das Model zu übertragen und umgekehrt. Diesen Vorgang des Datenaustausches bezeichnet man als „Data Binding“. [Wikipedia 4]

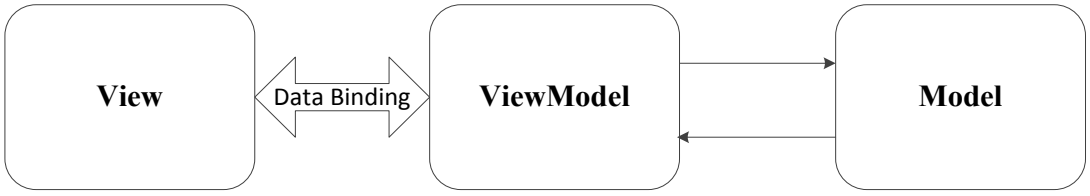

Abbildung 4: MVVM Architektur (vgl. [Wikipedia 5])

Abbildung 4 verdeutlicht die lose Kopplung zwischen Model und View. So ist es einfacher möglich, eine View für eine andere Anwendung wiederzuverwenden. Ein weiterer Vorteil dieser Vorgehensweise ist unter anderen, dass GUI und Logik unabhängig voneinander entwickelt werden können.

2.6 REST – Representational State Transfer

Representational State Transfer ist eine Beschreibung für die webgerechte Übertragung von Daten bzw. Ressourcen. Anfragen über dieses Protokoll werden zustandslos über das HTTP-Protokoll getätigt. Alle Ressourcen sind über die ihm zugeordnete URI abrufbar. In welcher detaillierten Form Dokumente übertragen werden, ist nicht festgelegt. [Bayer 2002, S.2ff]

REST umfasst die wichtigsten HTTP-Methoden des HTTP-Protokolls. Eine entscheidende Rolle spielen die CRUD[9]-Operationen. Dazu zählen GET, POST, PUT und DELETE. GET lädt die aufgerufene Ressource. POST fügt eine neue Ressource in eine vorhandene ein oder erzeugt sie. PUT ähnelt POST, der Unterschied besteht darin, dass PUT definitiv eine neue Ressource erzeugt oder sie ersetzt. Eine Ressource kann mithilfe von DELETE gelöscht werden. [Bayer 2002, S. 3]

Es existieren auch noch weitere dieser sogenannten „Verben“, die jedoch weniger häufig in der Praxis verwendet werden. Das sind HEAD zum Abfragen der Metadaten einer Ressource. Sowie OPTIONS für Informationen über die Methoden, die auf eine Ressource angewendet werden können. Die in der Praxis sehr selten vorkommenden CONNECT und TRACE bieten Aufschlüsse über die verwendete Verbindung. [Tilkov 2011, S. 51-56]

Zu beachten ist, dass es sich hierbei nur um eine Beschreibung einer Architektur handelt und nicht um eine konkrete Umsetzung. REST-Implementierungen, die das Konzept korrekt umsetzen, werden als RESTful bezeichnet. [Tilkov 2011, S. 10f]

[9] Create Read Update Delete

3 AngularJS

3.1 Allgemeines

AngularJS[10] ist ein von Google entwickeltes Open-Source JavaScript-Framework zur Entwicklung von dynamischen Webapplikationen auf Basis einer Single Page Architektur. Das Framework ist auf Modularität und Testbarkeit ausgelegt. [AngluarJS.de-EG] Infolgedessen ermöglicht es moderne Entwicklungsprozesse, wie zum Beispiel die testgetriebene Entwicklung.

Eine Besonderheit ist die Architektur des Frameworks. So verwendet AngularJS weder eine reine MVC, noch eine reine MVVM Implementation, sondern kombiniert beide. Grundlage bildet dabei das Model View Controller Pattern. Um das Konzept der Zwei-Wege-Datenbindung zu erreichen, nähert es sich dem MVVM-Muster an. [JavaScript heute, S. 70] Dieses Konstrukt wird in der Literatur auch als MV*-Muster bezeichnet.

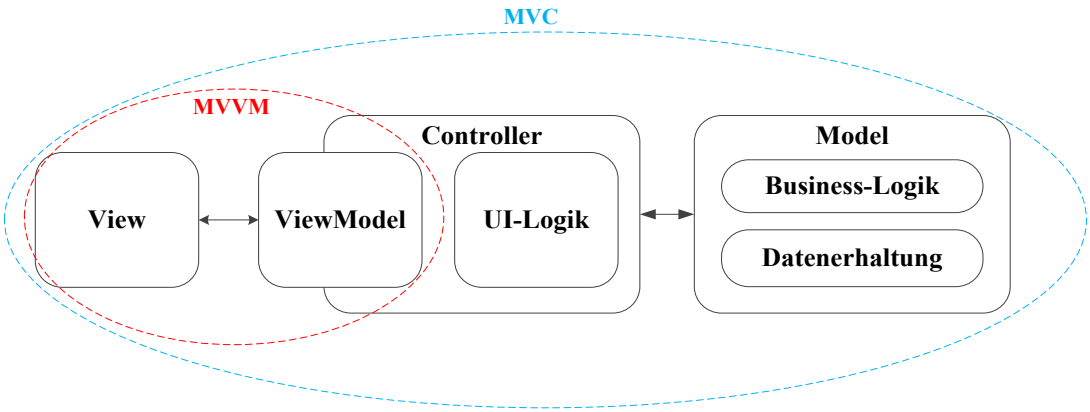

Abbildung 5: Kombination MVC und MVVM (vgl. [JavaScript heute, S. 71])

Diese **Zwei-Wege-Datenbindung** (Data Binding[11]) vereinfacht die Entwicklung von sich automatisch aktualisierenden Oberflächen. Wird eine Variable im Model geändert, so ändert sie sich auch in der View, ohne dass ein zusätzliches Event ausgelöst werden muss. Auch wenn der Benutzer etwas in ein Eingabefeld einträgt, wird der Wert der gebundenen Variablen geändert. Die Datenbindung wird über dem

[10] https://angularjs.org/
[11] Siehe Kapitel 2.5.2

Scope[12] realisiert und entkoppelt die View von der dahinter liegenden Logik. [Tarasiewcz & Böhm, S. 23] Abbildung 6 veranschaulicht dieses Prinzip.

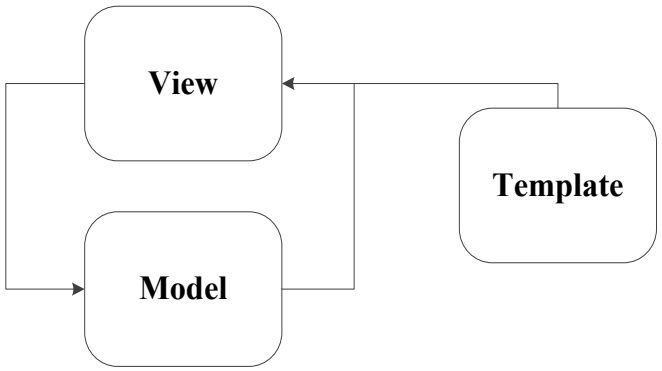

Abbildung 6: Schema der Zwei-Wege-Datenbindung (vgl. [AngularJS.de-DB])

Eine zentrale Rolle spielen dabei die **Templates**. Das sind HTML-Vorlagen, aus denen AngularJS die View generiert. Sie beinhalten neben HTML-Code auch Direktiven[13] und „Expressions“. Mit diesen **Expressions** ist es möglich die Daten des Scopes[14] anzubinden. Dies geschieht mithilfe des Variablen- oder Funktionsnamens, der in doppelt geschweifte Klammern „gepackt“ wird. [Tarasiewcz & Böhm, S. 33-35]

```
1  <div>
2    <p>
3      Hallo {{name}}
4    </p>
5  </div>
```

Listing 1: Beispiel einer Template mit einer Expression

Listing 1 zeigt, wie ein Template in AngularJS aussehen könnte. Die Expression {{name}} wird durch AngularJS mit dem Inhalt der in der Scope liegenden Variable „name“ durch den AngularJS-Compiler ersetzt. Das geschieht, aufgrund der bidirektionalen Datenverbindung, jedes Mal wenn sich dieser Wert ändert. Somit ist sichergestellt, dass die View immer aktuell gehalten wird.

[12] Siehe Kapitel 3.1
[13] Siehe Kapitel 3.4
[14] Siehe Kapitel 3.1

Um eine Modularisierung zu erreichen, verfügt AngularJS über einen Injektor-Mechanismus[15]. So ist es möglich Services, Direktiven, Controller, usw. nur mithilfe eines Bezeichners einzubinden. Der Injektor lädt dann die in der Konstruktorfunktion definierten Abhängigkeiten beim Laden der Anwendung. [AngularJS-DI]

Nicht vergessen werden darf, dass die Bibliothek in das HTML-Dokument geladen werden muss. Dafür muss sie (die Datei angular.js oder angular.min.js) über das <script> Tag eingebunden werden. Listing 2 zeigt das minimal notwendige Grundgerüst.

```
1  <!DOCTYPE html>
2  <html>
3  <head lang="de">
4     <meta charset="UTF-8">
5     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js">
6     </script>
7  </head>
8  <body ng-app>
9     Hier den Applikationscode einsetzen
10 </body>
11 </html>
```

Listing 2: Minimal notwendiges Grundgerüst einer AngularJS-Applikation

Anschließend muss das System noch wissen, in welchem Teil des Dokuments AngularJS geladen werden soll. Das geschieht über das spezielle HTML-Attribut (eine sogenannte AngularJS Direktive[16]) ng-app (Listing 2, Zeile 8). An welchem HTML-Tag dieses Attribut gehängt wird, spielt technisch gesehen keine Rolle. Der Entwickler kann somit entscheiden, ob AngularJS die gesamte Webseite steuert oder nur einen Teil von ihr. Insofern ist es theoretisch auch möglich, eine AngularJS-Anwendung in eine bestehende Webseite als Subsystem einzubinden.

[15] Dependency Injektion (siehe Kapitel 2.3.1)
[16] Siehe Kapitel 3.5

3.2 Scopes

Ein Scope referenziert einen Teil des Document Object Models und stellt das ViewModel der Anwendung dar. Demnach ist ein Scope ein elementarer Bestandteil der Zwei-Wege-Datenbindung. Variablen und Funktionen, die im Zusammenhang mit der View verwendet werden sollen, müssen in einem Scope abgelegt sein. Eine AngularJS Anwendung hat mindestens einen Scope, den „Root-Scope“. [Tarasiewcz & Böhm, S. 23]

Innerhalb einer AngularJS Anwendung kann es beliebig viele (ineinander verschachtelte) Scopes geben. So wird z.B. mit einem Controller auch ein dazugehöriger Scope erzeugt. Diese können auch voneinander erben. Es entsteht eine Hierarchie, beginnend mit den Root-Scope. Darüber ist es möglich, von einem Kind-Scope auf einem Eltern-Scope zuzugreifen. [Tarasiewcz & Böhm, S. 24]

Ausnahme hierbei bilden die **isolierten Scopes**. Diese können mithilfe von Direktiven[17] erzeugt werden. Sie ermöglichen dem Entwickler genau zu steuern, welche Variablen und Methoden vom Eltern-Scope verwendet werden. Es ist auch denkbar einen Zugriff auf den übergeordneten Scope komplett zu verbieten und den Kind-Scope damit komplett zu isolieren. [Tarasiewcz & Böhm, S. 24]

Scopes können miteinander kommunizieren. Folglich ist es vorstellbar Nachrichten über einen Broadcast zu Scopes zu versenden. Es gibt die Möglichkeit eine Nachricht an alle untergeordneten Scopes zu verschicken. Dies geschieht mithilfe von `$broadcast`[18]. Alternativ gibt es `$emit`[19], welches eine Nachricht an alle Übergeordneten sendet. [JavaScript heute, S. 72]

Innerhalb von Angular kann der lokale Scope über die variable `$scope`[20] abrufen werden. Der Root-Scope ist global über `$rootScope`[21] erreichbar.

[17] Siehe Kapitel 3.5
[18] https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$broadcast
[19] https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$emit
[20] https://docs.angularjs.org/api/ng/type/$rootScope.Scope
[21] https://docs.angularjs.org/api/ng/service/$rootScope

3.3 Module

In AngularJS werden Anwendungsteile, wie z.B. Controller, Services und Direktiven, in Module[22] gekapselt. Jede Anwendung hat mindestens ein Modul, also ist die Anwendung als solche ein „Modul“. Innerhalb eines solchen Moduls können Abhängigkeiten definiert und somit eine Verzweigung erzeugt werden. [Tarasiewcz & Böhm, S. 29]

Listing 3 zeigt, wie ein Entwickler mit Modulen arbeiten kann. So kann man ein Modul ohne Abhängigkeiten erstellen. Dafür wird einfach ein leeres Array als 2. Parameter übergeben (Listing 3, Zeile 2). Öfters werden jedoch weitere Submodule benötigt. So wird hier ein Modul mit dem Namen „Modul2“ erstellt. Dieses Modul bekommt nun ein Array mit 2 Einträgen übergeben (Listing 3, Zeile 5). Das heißt, „Modul1“ und „SubModul2“ werden in „Modul2“ geladen und somit innerhalb des ladenden Moduls zur Verfügung gestellt. Soll auf ein bereits definiertes Modul zugegriffen werden, so muss als 1. Parameter der Bezeichner übergeben und es wird kein 2. Parameter übergeben (Listing 3, Zeile 8).

```
1  //neues Modul ohne Abhängigkeiten erstellen
2  angular.module('Modul1', []);
3
4  //neues Modul mit Abhängigkeiten erstellen
5  angular.module('Modul2', ['Modul1', 'SubModul2']);
6
7  //auf vorhandenes Modul zugreifen
8  angular.module('Modul2');
```

Listing 3: Definition von AngularJS Modulen

Module bieten die Möglichkeit Konstanten zu erstellen. Dies geschieht über die Funktion .constant(name, value). Auch ist es vorstellbar, mit .conf() Voreinstellungen für Module zu treffen (wie z.B. das Routing[23]). Initiale Ausführungen können daraufhin innerhalb des .run() Blocks eines Moduls ausgeführt werden.

Injiziert der Entwickler ein Modul, so kann er auf alle sich darin befindenden Anwendungsteile zugreifen. In AngularJS ist es leider nicht möglich zu definieren,

[22] https://docs.angularjs.org/api/ng/type/angular.Module
[23] Zuordnung der Webpfade zu den entsprechenden Oberflächen

welche Anwendungsbausteine privat sind und welche nicht. Es ist also denkbar auf einen Service, welcher zur internen Verwendung konzipiert wurde, zuzugreifen. Das stellt einen Verstoß gegen eines der Grundprinzipien der Modularisierung dar. Um dieses Problem zu umgehen, müssen die Entwickler in der Dokumentation genau definieren, auf welche Bausteine eines Moduls zugegriffen werden darf.

3.4 Services und Controller

Ein elementarer Bestandteil eines jeden AngularJS Moduls sind „Controller". Daneben sollte jede Anwendung ergänzend „Services" beinhalten. Definiert werden sie, ähnlich wie ein Modul, über eine Service- bzw. Controllerfunktion. Beide können über Dependency Injection Abhängigkeiten laden. So ist es z.B. möglich, dass ein Controller einen Service verwendet.

Services kapseln komplexere Routinen und übergeordnete Daten. Empfehlenswert ist, dass der Großteil der clientseitigen Businesslogik in solche Services ausgelagert wird. Da es sich bei Services um *Singletons* handelt, sind deren Abläufe zentral verfügbar. [Tarasiewcz & Böhm, S. 42]

Es gibt viele Möglichkeiten einen Service in AngularJS zu definieren. So kann z.B. ein bereits vorhandenes JavaScript-Objekt mithilfe der Funktion .service(*object*) zu einem Angular-Service gemacht werden. Die am häufigsten verwendete Variante ist jedoch die .factory(…) Funktion. [Tarasiewcz & Böhm, S. 43f]

Listing 4 (Seite 24) zeigt beispielhaft die Definition eines Services mit dem Namen „ResourceService", der über eine Factory erzeugt wird. Dieser Service ist dann im Modul „meinModul" verfügbar. Der hier gebaute Service injiziert den Angular-Service $http. Dieser wird verwendet, um HTTP-Anfragen an einen Server zu senden.

```
1  angular.module('meinModul').factory('ResourceService',['$http',  function ($http) {
2    //Serviceinterne Funktion
3    function getResource() {
4        return $http.get(/*[...]*/);
5    }
6    /*[...]*/
7
8    //Definition der Schnittstelle
9    return {
10     getResource:  getResource;
11   }
12     /*[...]*/
13   }
14 }]);
```

Listing 4: Beispieldefinition eines Services

Im Gegensatz zur Moduldefinition kann der Entwickler bei der Service-Implementierung eine genau festgelegte Schnittstelle anlegen (Listing 4, Zeile 9-11). Somit kann genau festgelegt werden, welche Funktionen der Aufrufer des Services verwenden kann und welche nicht.

Controller sind für die UI-Logik verantwortlich. Sie agieren mit Scope-Variablen und verwalten das Event-Handling der Benutzeroberfläche. Services werden häufig von Controllern verwendet um den Scope zu befüllen. [Tarasiewcz & Böhm, S. 30]

```
1  angular.module('meinModul').controller('ViewCtrl', ['$scope', 'ResourceService',
2      function ($scope, ResourceService) {
3          $scope.anzeige = ResourceService.getResource();
4  }]);
```

Listing 5: Beispieldefinition eines Controllers

Listing 5 stellt die Entwicklung eines solchen Controllers dar. Hier wird im Modul „meinModul“ der Controller „ViewCtrl“ erstellt. Dieser Controller verwendet den im Listing 4 (siehe Seite 24) erstellten Service und lädt den Inhalt des Aufrufes in die Scope-Variable „anzeige“ (Listing 5, Zeile 3).

Ein Controller kann auf verschiedene Arten an eine View gebunden werden, z.B. über die Konfiguration im conf() Block des Moduls. Alternativ ist eine direkte Definition innerhalb eines Templates möglich. Dies geschieht mithilfe der Angular-Direktive „ng-controller“ (Listing 6, Seite 25).

```
1  <section ng-controller="ViewCtrl">
2    <p> {{anzeige}} </p>
3  </section>
```

Listing 6: Template die den ViewCtrl verwendet

3.5 Direktiven

Das Besondere an AngularJS ist, dass das Framework die Möglichkeit bietet, eigene HTML-Tags bzw. Attribute zu definieren. Realisiert wird dies mithilfe der sogenannten **Direktiven**[24]. Mit ihnen ist es denkbar, komplette Teilabschnitte der Anwendung (z.B. ein Formular) zu kapseln. Es ist auch möglich, nur DOM-Manipulationen wiederverwendbar zu programmieren. Eine Direktive wird wie Services und Controller an ein Modul gebunden. [Tarasiewcz & Böhm, S. 49f]

Direktiven werden mithilfe der .directive(...) Funktion erstellt und bieten eine ganze Reihe von Einstellungsmöglichkeiten. Eine Übersicht über eine Auswahl der wichtigsten Eigenschaften bietet Listing 7. Erwartet wird die Rückgabe eines Objektes zur Definition der Direktive, in der die Konfigurationen implementiert sind (Listing 7, Zeile: 2-14).

```
1  angular.module('meinModul').directive('meineDirektive', function() {
2    return {
3      restrict: 'E',
4      template: '<div><p>Hallo Direktive</p></div>',
5      link: function(scope, element, attrs) {
6        /*[...]*/
7      },
8      scope: false,
9      controller: function() {
10       /*[...]*/
11     },
12
13     /*[...]*/
14   }
15 });
```

Listing 7: Auswahl von Einstellmöglichkeiten des Direktiven Objektes

[24] https://code.angularjs.org/1.2.19/docs/guide/directive

Die „restrict“-Eigenschaft erwartet die Übergabe eines „Strings“ bzw. eines Zeichens. Hierbei wird gesteuert, wie ein Entwickler die Direktive in ein HTML-Dokument „einbauen“ kann. So ist in Listing 7 (Zeile 3) ein „E“ gesetzt, wodurch es nur möglich ist, die Direktive als HTML-Element einzubinden. Anzumerken ist, dass in der HTML-Seite die Benennung eine spezielle Konvention hat. So wird aus dem Direktiven-Namen „meineDirektive“ innerhalb des nutzenden HTML-Dokumentes <meine-direktive>…</meine-direktive>. Kurz gesagt wird aus einem Großbuchstaben, ein Kleinbuchstabe mit „–“ Präfix. Alternativ wäre statt dem „–“, auch ein „:“ oder „_“ denkbar. Weitere „restrict“-Einstellungen können der Tabelle 1 entnommen werden.

Zeichen	Einbindung über	Beispiel
E	Element	<meine-direktive></meine-direktive>
A	Attribut	<div meine-direktive>
C	CSS-Klasse	
M	Kommentar	<!--meine-direktive-->

Tabelle 1: Mögliche „restrict“ Deklarationen (vgl. [Green & Seshadri, S. 122])

Diese Vorschriften können beliebig miteinander kombiniert werden. So kann auch ein „AE“ übergeben werden. Dann wäre es möglich, die Direktive als HTML-Element oder als Attribut eines HTML-Elementes einzubinden.

An jede Direktive kann auch ein Template gebunden werden. Demonstriert wird dies in Zeile 4 im Listing 7 (Seite 25). Überall dort, wo die Direktive in Gebrauch ist, bindet AnuglarJS dieses definierte Template ein. Das Benutzen von „Expressions“ und anderen AngularJS HTML-Elementen innerhalb eines solchen Templates ist ebenfalls denkbar.

Alternativ kann statt einer direkten Template-Definition auch der Pfad zu einem HTML-Dokument angegeben werden. Dafür muss jedoch die Eigenschaft „template“ durch „templateUrl“ ersetzt werden.

Eine nicht unwesentliche Rolle spielt die „link“-Eigenschaft (Listing 7, Zeile 5). Die darin definierte Funktion wird von jeder Direktiven-Instanz einmal aufgerufen. Diese Funktion eignet sich daher hervorragend zur Kapselung von Manipulationen auf dem

Scope und dem DOM. Die Funktion bekommt 3 Übergabeparameter mitgeliefert. Dies sind: „scope“ – der Scope dieser Direktive, „element“ – das HTML-Element, welches die Direktive verwendet und „attrs“ – die an die Direktive übergebenen Attribute.

Sollte eine Direktive sehr groß werden, kann der Entwickler sich dazu entscheiden, einen eigenen Controller zu definieren (Listing 7, Zeile 9). So kann eine Funktion übergeben werden, die alle Controller-Funktionalitäten beinhaltet. Alternativ ist es auch möglich einen AngularJS-Controller, wie in Kapitel 3.4 beschrieben, zu definieren und diesen per Dependency Injection einzubinden.

Direktiven bieten die Möglichkeit, den Scope detailliert einzustellen. In Listing 7, Zeile 8, wird der Wert „false“ übergeben. Das bewirkt, dass die Direktive keinen eigenen Scope bekommt und der übergeordnete Scope mitverwendet wird. Alternativ wäre „true“, zur Erzeugung eines eigenen (öffentlichen) Scopes, einsetzbar.

Interessant ist der Fall der Erzeugung eines **isolierten Scope**[25]. Dieser wird durch die Übergabe eines Objektes erzeugt. Innerhalb dieses Objektes kann einfach über Zeichen als Eigenschaftswert definiert werden, welche Attribute und Funktionen über den Scope nach außen bzw. innen gekoppelt sind. Ein Leerobjekt hätte zur Folge, dass die Direktive komplett abgeschottet ist.

```
1  /*[...]*/
2  scope: {
3     title: '@',
4     date: '=',
5     onClose: '&'
6  }
7  /*[...]*/
```

Listing 8: Erzeugung einer isolierten Scope (Ausschnitt aus einer Direktive)

In Listing 8 wird eine Direktive mit isoliertem Scope implementiert. Insgesamt umfasst sie 3 Attribute. In Zeile 3 wird eine Ein-Wege-Datenbindung durch den Wert „@“ in die Direktive gelegt. Das heißt „title“ wird nur von außen übergeben und Änderungen, innerhalb des geschlossenen Scopes, haben keinen Einfluss auf den

[25] Siehe Kapitel 3.2

äußeren Wert der Variable. Durch „=“ wird in Zeile 4 eine Zwei-Wege-Datenbindung zwischen inneren und äußeren Scope auf die Variable „date“ gesetzt. Als Letztes erfolgt die Übergabe einer Funktion über das Attribut „onClose“. Dafür muss als Eigenschaftswert das Zeichen „&“ übergeben werden.

3.6 Umgang mit Asynchronität

Eine JavaScript-Anwendung selbst besteht aus nur einem einzigen Thread. Trotzdem bietet JavaScript eine Reihe von asynchronen Mechanismen bzw. Aufrufen, wie z.B. den „HTTP-Request“, an. Diese asynchronen Events werden in der sogenannten „Event-Loop“ eingereiht und abgearbeitet. [JavaScript.Info]

Um nun das System nicht zu blockieren, übernimmt die JavaScript-Laufzeitumgebung die eigentliche Abarbeitung des Event-Loops, wie z.B. das Laden der Daten vom Server. Die Laufzeitumgebung prüft in der Ereignisschleife regelmäßig den Stand der Abarbeitung und benachrichtigt den entsprechenden JavaScript-Thread über das Ende der Abarbeitung. Code, der nun zu einem verzögerten Zeitpunkt abgearbeitet werden soll, wird über einen sogenannten „Callback“ an die asynchrone Funktion übergeben. Diese führt den Callback, z.B. nach einem Timeout, aus. [Swenson-Healey 2013]

Problematisch sind insbesondere mehrere voneinander abhängige asynchrone Aufrufe. Mithilfe von Callbacks kann dieses Problem nur durch eine Verschachtelung behoben werden. Mit zunehmender Zahl von Abhängigkeiten steigt auch die Tiefe der Struktur. Es entsteht ein Konstrukt, welches als „Pyramid of Doom“ (Listing 9, Seite 29) bezeichnet wird. [Burnham 2012, S. xii]

```
1  schritt1(function(ergebnis1) {
2    schritt2(function(erbenis2) {
3      schritt3(function(ergebnis3) {
4        schritt4(function(ergebnis4) {
5              /*[...]*/
6        });
7      });
8    }) ;
9  });
```

Listing 9: Pyramid of Doom (vgl. [Burnham 2012, S. xii])

Um dieses Problem zu umgehen, hat sich eine elegante Lösung etabliert. Die Callbacks werden durch „Promises“[26] ersetzt. Ein **Promise** ist eine Abstraktion eines zukünftig eintreffenden Ergebnisses. [Tarasiewcz & Böhm, S. 308] Einfacher formuliert handelt es sich um eine Art „Markierung“. Das System weiß, dass ein Promise zu einem späteren Zeitpunkt ein Ergebnis liefert, und ist somit in der Lage, das Ergebnis nachzuladen bzw. zu erhalten. Auch ein Abarbeiten von abhängigen asynchronen Aufrufen lässt sich sauber implementieren.

Zur Realisierung des Konzeptes der Promises liefert die AngularJS API den $q[27]-Service. Dieser Service besteht im Wesentlichen aus 2 Teilen, einem internen und einem externen Teil.

Der *interne* Bestandteil ist die Implementierung der Funktion des asynchronen Aufrufes. Sie beinhaltet ein sogenanntes „Defer“-Objekt, welches für die Aussteuerung des Zustandes des asynchronen Aufrufes verantwortlich ist. Dieses Objekt gibt außerdem das eigentliche Promise zurück. Dieses Promise ist der *externe* Teil, welcher von der aufrufenden Funktion verwendet, wird.

Über *defer* = $q.defer() wird ein Defer-Objekt erzeugt und mit *defer*.promise wird das entsprechende Promise zurückgegeben. Das Defer-Objekt kann dann innerhalb einer asynchronen Funktion mit *defer*.resolve(*ergebnisObj*) aufgelöst oder mit *defer*.reject(*fehlerObj*) abgelehnt werden. Mit dem über das Defer-Objekt erzeugten Promise wird dann die Weiterverbreitung nach dem asynchronen Aufruf geregelt.

[26] auch als „Futures“ bezeichnet
[27] https://code.angularjs.org/1.2.19/docs/api/ng/service/$q

Dies geschieht z.B. über *promise*.then(*erfolgsFunktion*, *fehlerFunktion*). Folgend soll ein kleines Beispiel die Promise-API veranschaulichen.

In Listing 10 wird zunächst von Zeile 1-13 die asynchrone Funktion implementiert. Hier im Beispiel wird einfach 500ms gewartet. Denkbar wäre es auch, dass stattdessen eine HTTP-Anfrage läuft.

```
1  function asyncFunction(condition) {
2    var defered = $q.defer();
3
4     setTimeout(function() {
5        if(condition) {
6           defered.resolve(condition);
7        } else {
8           defered.reject(condition);
9        }
10    }, 500);
11
12    return defered.promise;
13 }
14
15 function usingFunction() {
16    var promise = asyncFunction(true);
17
18    promise.then(
19       function(response) {
20          console.log("Alles gut!: "+ response);
21       },
22       function(response) {
23          console.log("Fehler!: "+ response+ "nicht eingetroffen");
24       })
25       .then(function() {
26          console.log("Alles abgearbeitet");
27    });
28 }
```

Listing 10: Beispiel Promise-API

Es gibt eine zweite aufrufende Funktion (Listing 10, Zeile 15-28). Dort wird gesteuert, was im Erfolgs- bzw. Fehlerfall passiert. Im Beispiel wird einfach nur ein Text und der Wert der asynchronen Funktion in die Konsole geschrieben. Um genau zu sein muss in diesen Fall, nach frühestens 500ms, die Ausgabe „Alles gut!: true“ zu sehen sein. Nachdem dieses „then(...)“ (Listing 10, Zeile 18-24) abgearbeitet ist, folgt die nächste Durchführung (Listing 10, Zeile 25-27). Darauf wird also auch noch immer „Alles abgearbeitet“ ausgegeben. So ließe sich dementsprechend auch eine

Vielzahl von abhängigen asynchronen Aufrufen, mithilfe von „then(...)“, übersichtlich verwerten.

3.7 Anbindungen an Backendsysteme

Für die Kommunikation mit einem Server realisiert der Browser das Ajax-Programmiermodell[28] mithilfe des XMLHttpRequest-Objekts. Dadurch ist der Client in der Lage, Inhalte dynamisch asynchron nachzuladen. Um diesen Mechanismus zu vereinfachen und die Promise-API[29] zu nutzen, bietet AngularJS hierfür einen eigenen $http[30]-Service an. [Tarasiewcz & Böhm, S. 213f]

Da dieser auf dem HTTP-Protokoll basiert, lässt sich über den $http-Service auch leicht ein REST-Webservice anbinden. Für diesen Zweck bietet der $http extra eine Reihe von Funktionen an. So werden HTTP- bzw. REST-Methoden, wie GET und POST, mit $http.get(*url*) bzw. $http.post(*url, data*) realisiert.

Möchte ein Entwickler detailliertere Einstellungen treffen, so kann er dies ohne Weiteres tun. Dafür muss er lediglich ein Konfigurationsobjekt an den $http-Service übergeben. Dieses Konfigurationsobjekt bietet z.B. die Möglichkeit, den HTTP-Header einzustellen oder eine Timeout-Zeit festzulegen. Wenn gewünscht, lässt sich auch der gesamte HTTP-Aufruf darüber steuern. Dieses Konfigurationsobjekt wird einfach als Funktionsparameter direkt an den $http-Service gesetzt: $http(*confObj*). Das Beispiel in Listing 11 (Seite 32) veranschaulicht die Verwendung.

Listing 11 (Seite 32) zeigt den Aufruf eines REST-Service, der sich auf einem „Server“ im lokalen Umfeld befindet. Als Erstes wird in Zeile 3 eine POST-Anfrage an diesen Server abgesetzt. Auf eine Abarbeitung der Serverantwort wird an dieser Stelle aus Übersichtsgründen verzichtet.

[28] Siehe Kapitel 2.2
[29] Siehe Kapitel 3.6
[30] https://code.angularjs.org/1.2.19/docs/api/ng/service/$http

```
1  var userdata = {name: "max", pw: "1234"};
2  //Login
3  $http.post('http://localhost:8080/api/login', userdata);
4
5  //Informationen Abfragen
6  $http({
7     method: 'GET',
8     url: 'http://localhost:8080/api/info',
9     headers: {Charset: 'utf-8'}
10 }).then(function(response) {
11    $scope.info = response.data;
12    console.log(response.status);
13 });
```

Listing 11: Beispiel $http-Service

Danach wird ab Zeile 6 ein GET-Aufruf getätigt. Hier wird zur Veranschaulichung ein Konfigurationsobjekt verwendet. Jetzt muss selbstverständlich auf die Antwort des Servers reagiert werden. Da der $http-Service die Promise-API implementiert, gibt der Service ein Promise-Objekt zurück. Dieses kann einfach, wie in Kapitel 3.5 beschrieben, verwendet werden. Zusätzlich zu der bekannten .then(..) Funktion stehen noch .success(...) und .error(...) zur Verfügung. Diese können alternativ oder ergänzend verwendet werden.

Ist das Promise aufgelöst, steht ein Response-Objekt (Listing 11, Zeile 10) zur Verfügung. Über dieses Objekt lassen sich die eigentlichen Daten der Antwort, sowie zusätzliche Informationen, abrufen. Im Beispiel wird in den HTTP-Body (die Daten) die Antwort in die Scope-Variable „info“ und anschließend der HTTP-Status in der Browserkonsole ausgegeben (Listing 11, Zeile 11-12).

3.8 Tests

Ein wichtiger Aspekt der Softwareentwicklung ist das Testen von Anwendungen bzw. Anwendungsbestandteilen. Das AngularJS-Framework bietet dafür die Möglichkeit Unit- und E2E(End-to-End)- Tests zu schreiben. Um diese Tests jedoch schreiben und ausführen zu können, werden zusätzlich ein „Testrunner“[31] und ein Test-Framework benötigt. Viel verwendet sind zum Beispiel der Testrunner „Karma“[32] und das Framework „Jasmine“[33], die unter anderem auch im Rahmen der Arbeit benutzt werden sollen.

Unit-Tests umfassen die Testfälle für einzelne Anwendungsbausteine, wie z.B. Services, Controller oder Direktiven. Bei solchen Tests werden die Bestandteile einzeln und unabhängig voneinander getestet, was einem sogenannten Whitebox-Test entspricht. Dadurch soll gewährleistet werden, dass sämtliche Methoden richtig funktionieren. Das Zusammenspiel von Systemen spielt dabei eine untergeordnete Rolle. [Vermeersch 2014]

Bei **End-to-End-Tests** wird die Anwendung als Blackbox betrachtet. Ziel ist es jetzt, die gesamte oder große zusammenhängende Teile der Anwendung möglichst als Ganzes zu testen. Die Benutzerinteraktion und Zusammenhänge stehen im Vordergrund. [Tarasiewcz & Böhm, S. 25f] Mithilfe des vom AngularJS mitgelieferten Test-Framework „Protractor“[34] können Interaktionen eines Benutzers mit einem Browser simuliert werden. Es ist auch denkbar, innerhalb eines Tests, viele verschiedene Browser vollautomatisch zu testen.

Auch wenn die Konzepte und die verwendeten Test-Frameworks verschieden sind, so ist der grundlegende Aufbau beider Testarten gleich. Ein Testfall besteht immer mindestens aus einer Ansammlung von einzelnen Test-Durchführungen samt Erwartungshaltungen. Abbildung 7 (Seite 34) veranschaulicht den Aufbau. Optional kann zusätzlich eine Test-Initialisierung und/oder abschließende Aufräumarbeiten der Testfälle stattfinden.

[31] Spezielle Laufzeitumgebung für Tests
[32] http://karma-runner.github.io/
[33] http://jasmine.github.io/
[34] https://github.com/angular/protractor

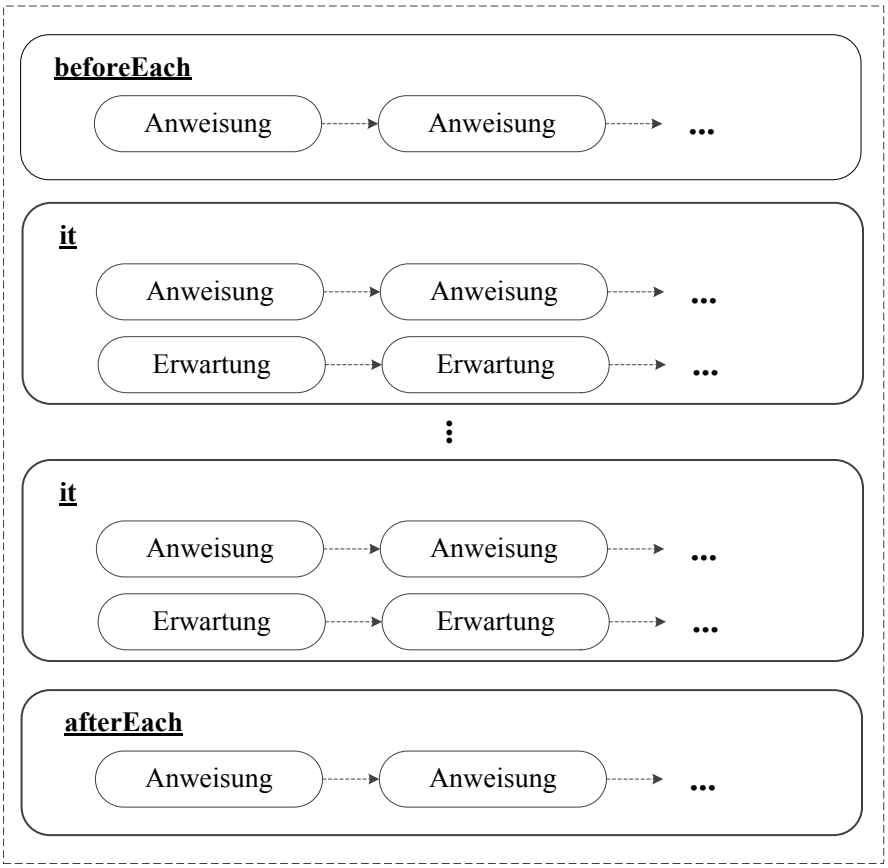

Abbildung 7: Aufbau eines Testfalls (vgl. [AngularJS-E2E])

Beispielsweise soll in Listing 12 (Seite 35) eine Funktion eines Services zum Prüfen des „größer als“ Verhältnisses zweier Zahlen getestet werden. Dafür wird mithilfe der Funktion describe(...) (Listing 12, Zeile 1-11) ein Testfall erstellt. Der zu testende Service befindet sich hier in dem Modul „myModule“. Deshalb muss das Modul in der Test-Initialisierung beforeEach(...) geladen werden (Listing 12, Zeile 2-4). Jetzt können einzelne Test-Durchführungen mithilfe der it(...) Funktion implementiert werden.

In diesem Beispiel wird der Service „myService“ aus dem geladenen Modul benötigt. Dafür muss der Service innerhalb der Durchführung injiziert werden. Der AngularJS Injektor-Mechanismus wird über die Funktion inject(...) (Listing 12, Zeile 6) ausgelöst. Abschließend ist es notwendig, die Erwartungen über expect(...) zu definieren. Im Beispiel wird geprüft, ob der Rückgabewert der Funktion „true“ ist (Listing 12, Zeile 9).

```
1  describe("Test-Beispiel", function () {
2    beforeEach(function () {
3      module('myModule');
4    });
5
6    it("Funktionsaufruf sollte gütlig sein", inject(['myService', function (myService) {
7      var bool = myService.isGreaterThen(1,4);
8
9      expect(bool).toBe(true);
10   }]));
11 })
```

Listing 12: Test-Beispiel

Auch wenn automatisierte Tests sehr wichtig und praktisch sind, so muss das Verhältnis von Nutzen und Aufwand stimmen. Das kann besonders bei E2E-Tests des User-Interfaces zum Problem werden. So lässt sich zwar mit ihnen in AngularJS (im Vergleich zu anderen Systemen) relativ leicht der UI-Workflow überprüfen, aber das Testen von Anzeigedetails, wie z.B. Abstände, wird schnell sehr aufwendig. Solche Anzeigedetails lassen sich nur mit erheblicher Mehrarbeit automatisch überprüfen. An dieser Stelle sollte überlegt werden, ob es Sinn ergibt, automatische Tests dafür zu schreiben. Oft ist es praktikabler, wenn ein Entwickler den Workflow automatisch testet, sich aber dann die Anzeige manuell anschaut. Letzten Endes ist die Entscheidung über das genaue Vorgehen an der Stelle nicht einfach und sollte von Fall zu Fall individuell getroffen werden.

4 Prototyp

4.1 Konzept

Die Reisezeiterfassungs-Applikation der GISA GmbH ist eine für den internen Gebrauch gedachte Webanwendung. Mitarbeiter der Firma sollen die Möglichkeit haben, ihre Dienstreisezeiten schnell und unkompliziert zu erfassen und alle Anträge anschließend automatisch zu stellen. Es wird zwischen Tagesreisen und Mehrtagesreisen unterschieden.

Die Zeiterfassung soll von einem Desktop-Rechner oder mobilen Endgerät aus getätigt werden können. Die Webapplikation wird prinzipiell auf allen Geräten laufen, die einen aktuellen Webbrowser besitzen.

Ein Großteil der Geschäftslogik (z.B. die Antragsstellungen) und die Speicherung der Daten übernimmt das Backend. Dieses Backend ist kein Bestandteil dieser Arbeit und wird als gegeben betrachtet.

Der Client leitet den Benutzer zu der auf der Startseite ausgewählten Eingabemaske. Dieses Eingabeformular wird entsprechend an den Typ der Reise angepasst. Trotz der speziellen Anpassungen setzen beide Eingabemasken auf eine gemeinsame Basis. Von diesen ausgehend geht es anschließend weiter zur Auswertung. Eine Rücknavigation soll jederzeit möglich sein. Abbildung 8 verdeutlicht die Navigation.

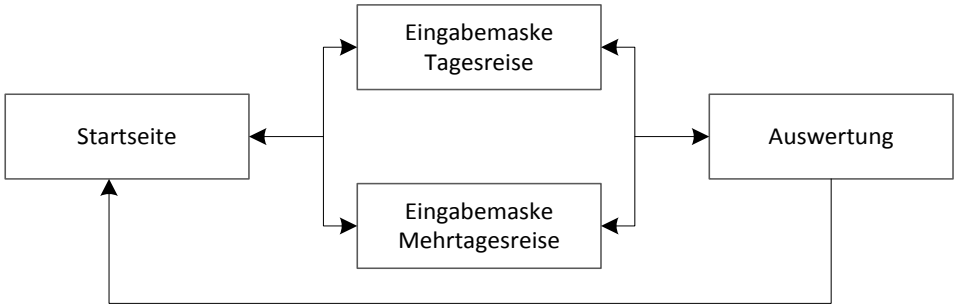

Abbildung 8: UI-Gesamtaufbau mit Navigationsmöglichkeiten

Der Client sendet die eingegebenen Daten der jeweiligen Maske an das Backend. Dieses System berechnet anschließend alle notwendigen Daten. Beim Aufruf der Auswertungsoberfläche lädt der Client die berechneten Daten vom Server und zeigt diese aufbereitet an. Der Benutzer hat nun die Wahl, ob er die Antragstellung in Gang setzt, oder die Daten nur zwischenspeichert. Beides übernimmt wieder das Backend. Dies arbeitet unabhängig vom Client, somit muss der Benutzer nicht warten, bis einer dieser Vorgänge vollständig abgeschlossen ist.

4.1.1 Startseite und Tagesreise

Die Startseite soll lediglich aus 2 Buttons bestehen. Der einzige Zweck, den diese View erfüllt, ist die Weiterleitung des Benutzers, zu den entsprechenden „Eingabe-Views“.

Die Oberfläche für die Eingabe einer Tagesreise ist eine simple Darstellung für die Eingabe von Datums- und Zeitangaben. Das User Interface besteht im Wesentlichen aus einer Datumseingabe und der Eingabemöglichkeit der Zeiträume. Diese Zeiträume sind „Hinfahrt“, „Arbeitszeit“ und „Rückfahrt“. Mehr Daten sind für die Erfassung einer Tagesreise nicht nötig. Abbildung 9 zeigt den konzeptionellen Aufbau der Oberfläche.

Abbildung 9: UI-Konzept für die Erfassung einer Tagesreise

4.1.2 Mehrtagesreise

Das User Interface für die Mehrtagesreise beinhaltet eine etwas komplexere Struktur, basiert jedoch auf den gleichen Elementen wie die Tagesreise. Innerhalb der Mehrtagesreise spielt die Eingabemöglichkeit mehrerer Arbeitstage eine hervorzuhebende Rolle.

Zusätzlich muss die Eingabe der Hinfahrt und Rückfahrt gewährleistet sein. Eingabefelder zur Erfassung von Arbeitstagen sollen über einen Button „Tag hinzufügen“ hinzugefügt werden können. Das Entfernen eines solchen Tages soll ebenfalls jederzeit möglich sein. Die folgende Grafik (Abbildung 10) illustriert das Konzept.

Abbildung 10: UI-Konzept der Mehrtagesreise

4.1.3 Auswertungsoberfläche

Abschließend soll die View „Auswertung“ eine Übersicht über die berechneten Daten zeigen. Auch hier wird zwischen Tages- und Mehrtagesreise unterschieden. Die Auswertung der Tagesreise beinhaltet eine einfache tabellarische Übersicht der relevanten Daten. Bei der Anzeige einer Mehrtagesreise ist der Aufbau simultan. Der zwingende Unterschied ist, dass jetzt eine Liste der einzelnen Tage dargelegt wird.

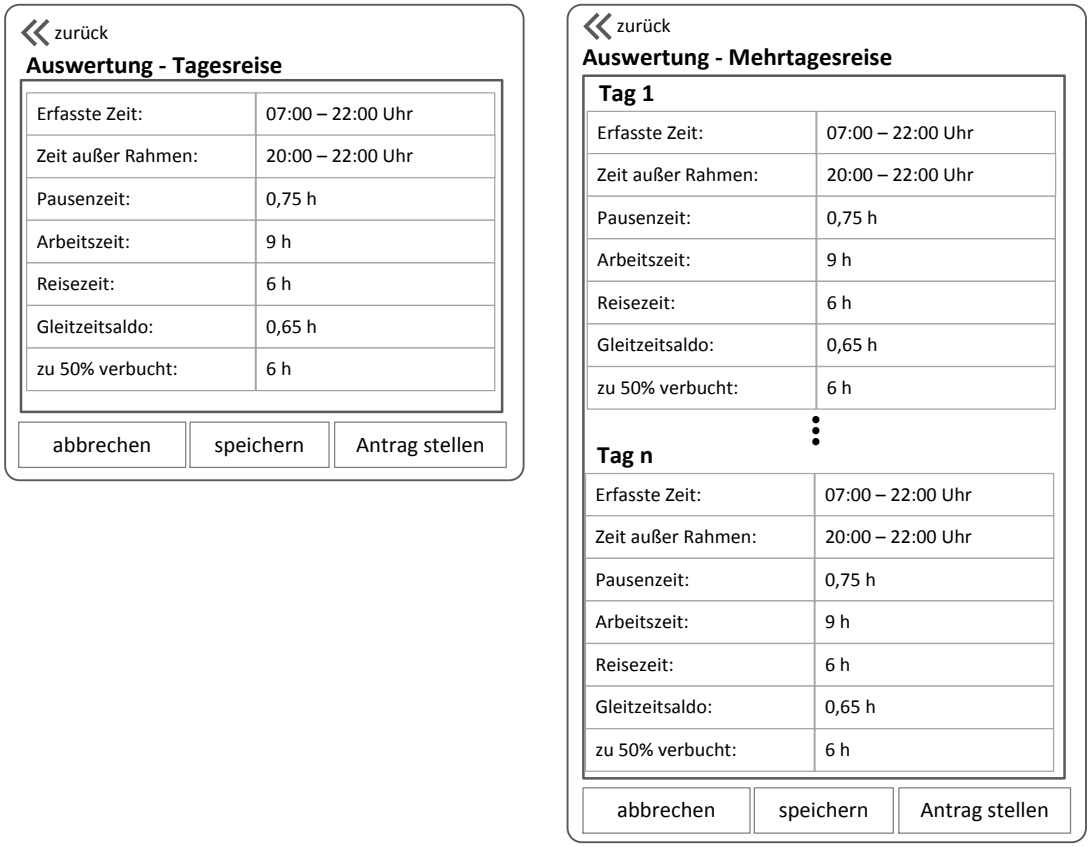

Abbildung 11: UI-Konzept für die Auswertung der Tagesreise (links) und Mehrtagesreise (rechts).

Die Auswertungsoberfläche, sowie alle anderen Views, müssen zur vollen Funktionalitätserfüllung mit dem Backend kommunizieren. Dafür soll ein zentraler Service zur Verfügung stehen, der den entsprechenden REST[35]-Webservice implementiert.

[35] Siehe Kapitel 2.6

4.2 Realisierung

Bei der Reisezeitenerfassung[36] handelt es sich um eine modulare AngularJS Webanwendung. Um allen Anforderungen gerecht zu werden, benötigt die Applikation noch weitere von AngularJS unabhängige Bibliotheken[37]. So vereinfacht zum Beispiel JQuery[38] die DOM-Manipulationen.

Als weitere Bibliothek wird unter anderem das CSS-Framework **Bootstrap**[39] verwendet. Dieses ermöglicht es, sogenannte „Responsive Webpages“ zu erzeugen. Das sind Seiten, bei denen sich die Oberfläche an die Fenstergröße des Browsers dynamisch anpasst. Die parallele Unterstützung von Mobilgeräten und Desktops wird dadurch stark erleichtert. So wird die Applikation nur ein gemeinsames Template für beide Gerätetypen implementieren.

Eine Auswahl an vorgefertigte UI-Widgets liefert die von AnuglarJS abhängige Bibliothek **AngularUI**[40]. Es handelt sich um eine Ansammlung von Bootstrap-Direktiven, wie z.B. einen „Timepicker“ oder einen „Datepicker“.

Um die Verwaltung dieser externen Libraries zu vereinfachen wird das Paketverwaltungstool namens **Bower**[41] Gebrauch gemacht. Das Tool ermöglicht es Pakete, Bibliotheken und Frameworks über eine zentrale Konfigurationsdatei als Abhängigkeiten zu definieren. Anschließend ist es möglich, sie einfach per Konsolenbefehl zu installieren. Ein eigenmächtiges Verteilen der Bibliotheken an alle Entwickler ist somit nicht mehr nötig, es muss lediglich die Bower-Konfigurationsdatei zu Verfügung gestellt werden.

An dieser Stelle ist erkennbar, dass zur Entwicklung von Webapplikationen eine Vielzahl von Tools und Frameworks erforderlich ist. Auf den ersten Blick handelt es sich dabei um einen kleinen Nachteil der modernen Webentwicklung. Ein Entwickler sollte im Idealfall über ein umfangreiches Wissen über alle eingesetzten Technologien verfügen. Da alle Frameworks jedoch auf den in Kapitel 2.2

[36] Die Anwendung mit allen dazugehörigen Dateien befindet sich auf der CD im Anhang
[37] Diese Bibliotheken werden im Rahmen der Arbeit nur kurz erwähnt und nicht näher analysiert.
[38] http://jquery.com/
[39] http://getbootstrap.com/
[40] http://angular-ui.github.io/
[41] http://bower.io/

erläuterten Techniken aufbauen, ist die Lernkurve sehr hoch und die Einarbeitung in neue Systeme stellt kein Problem dar.

4.2.1 Projektstruktur

Zentraler Ausgangspunkt der Anwendung ist die Datei „index.html“. Dieses HTML-Dokument beinhaltet alle Libraries, den Stylesheet und sämtliche verwendete JavaScript-Dateien (Module, Services, Controller usw.). Darin befindet sich auch die Festlegung, dass AngularJS den gesamten Inhalt (Body) des Dokumentes steuert.

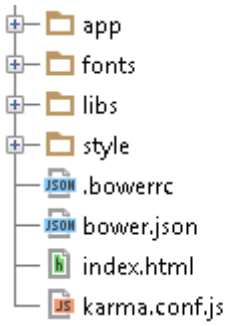

Abbildung 12: Projektstruktur

Abbildung 12 zeigt die Projektstruktur der Anwendung. Der Ordner „app“ enthält alle entwickelten Module. In „fonts“ befinden sich die verwendeten Schriftarten und „style“ enthält die Formatierungsvorlagen. Alle verwendeten Bibliotheken befinden sich im Ordner „libs“. Das Hauptverzeichnis birgt, neben der bereits erwähnten „index.html“, auch die Konfigurationsdateien für Bower und Karma[42].

Offensichtlich ist, dass die Webanwendung aus vielen einzelnen Dateien besteht. Einige davon sind sogar nur für die Entwicklung relevant und spielen für den Benutzer während der Ausführung keine Rolle. Es ist daher von Bedeutung, dass sich auf dem Webserver abschließend eine kompakte Version der Webapplikation befindet. Dafür sollte zur Livesetzung der Anwendung der geschrieben Quellcode zu einer Datei zusammengezogen und unnötige Zeichen (z.B. Kommentare) entfernt werden.

[42] Siehe Kapitel 3.8

Die Einbindung aller einzelnen JS-Dateien in der index.html stellt einen nicht zu unterschätzenden Nachteil dar. Fügt ein Entwickler ein neues Modul hinzu, darf er nicht vergessen, alle Dateien in die index.html zu setzen. Andernfalls kommt es zu Fehlermeldungen und die Anwendung wird in ihrem Funktionsumfang erheblich beschränkt. Im schlimmsten Fall ist die Applikation nicht mehr lauffähig.

4.2.2 Modularer Aufbau

Wie in Kapitel 3.3 beschrieben, handelt es sich bei der Anwendung selbst um ein Modul. Dieses Modul namens „gisa.app“ bildet die Wurzel des „Modul-Baums“. Dort werden alle untergeordneten Bestandteile eingebunden und alle globalen Einstellungen vorgenommen.

Kern der Applikation ist das fachliche Modul „gisa.travelTimeRegistration“. Dieser Anwendungsbestandteil selbst injiziert das AngularJS eigene Module „ngRoute“, um Routing-Mechanismen zu ermöglichen.

Daneben gibt es noch technische Hilfssysteme. So umfasst „gisa.utils“ alle allgemein verwendbare Services und Direktiven. Für sprachspezifische Ausgaben und Formatierungen wird „ngLocale“ zur Verfügung gestellt.

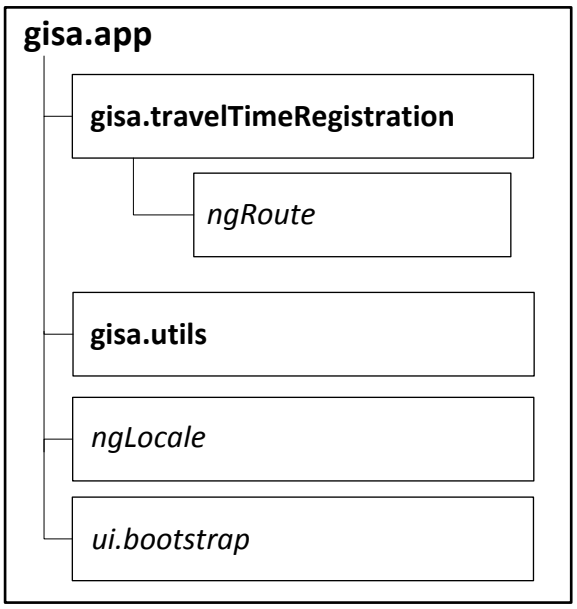

Abbildung 13: Modularer Aufbau der Reisezeitenerfassung

Abschließend bedient sich das Programm, über das Modul „ui.bootstrap“, an der AngularUI Bibliothek. Dieser Aufbau ist in Abbildung 13 (Seite 42) visualisiert. Eine Erweiterung um weitere Funktionalitäten ist vorstellbar. Zur Bewerkstelligung dieser Aufgabe muss einfach ein weiteres Modul in „gisa.app“ eingebunden werden.

Module, die sich in eigener Entwicklung befinden, folgen einem einheitlichen Schema (siehe Abbildung 14). Im Ordner „app“ des Quellcodes befindet sich jedes Modul innerhalb eines Ordners mit dem jeweiligen Modulnamen als Bezeichnung. Zentraler Ausgangspunkt eines jeden Moduls ist die JavaScript-Datei „*modulname*.js“, die sich im Wurzelverzeichnis befindet. In dieser wird das Modul definiert und alle lokalen Einstellungen getroffen. Alle weiteren Bestandteile befinden sich in Unterordner sortiert.

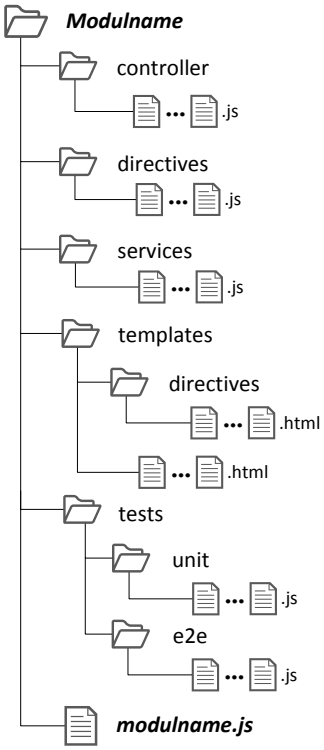

Abbildung 14: Modulschema

Insbesondere Wartbarkeit und Entwicklerfreundlichkeit werden durch dieses einheitliche und übersichtliche Muster gefördert. Da es keinen (funktionalen) Zwang zu einer solchen einheitlichen Grundform gibt, sollte während der Entstehung weiterer Module darauf geachtet werden, dass sich alle Entwickler daran halten.

Wird über einen empfohlenen Aufbau recherchiert, so finden sich durchaus auch leicht abgewandelte Formen. Ebenso werden die Tests gerne außerhalb des Moduls (als extra Ordner) untergebracht. Dadurch soll verhindert werden, dass die Tests bei der Veröffentlichung mit ausgeliefert werden. Ist die Trennung realisiert, muss jedoch darauf geachtet werden, dass beim Einfügen eines Moduls ebenfalls die Tests eingesetzt werden. In diesem Projekt sind die Tests jedoch modulintern definiert und werden bei einer automatisierten Publikation der Applikation einfach nicht mitgeliefert. Die hier verwendete Vorgehenswiese ist modularer.

4.2.3 Utils-Modul

Hierbei handelt es sich um ein technisches Modul zur Kapselung aller allgemein verwendbaren Elemente der Anwendung. Es erfüllt dementsprechend keine spezifische fachliche Aufgabe und kann in allen AngularJS-Applikationen wiederverwendet werden.

Von besonderer Bedeutung sind die hier definierten Direktiven. Bei allen handelt es sich um Kapselungen von UI-Widgets. Folgendermaßen wird der „Rahmen“ der Applikation in der Direktive „appFrame“ festgelegt. Ebendiese kapselt das „Panel“-Element aus AngularUI und passt es direkt an die Bedürfnisse eines Applikationsfensters an. Alle fachlichen Oberflächen werden in dieses Panel durch diese Direktive eingebunden. Überschrift und Aktionsfunktionen werden über Attribute, wie in Kapitel 3.5 beschrieben, übergeben.

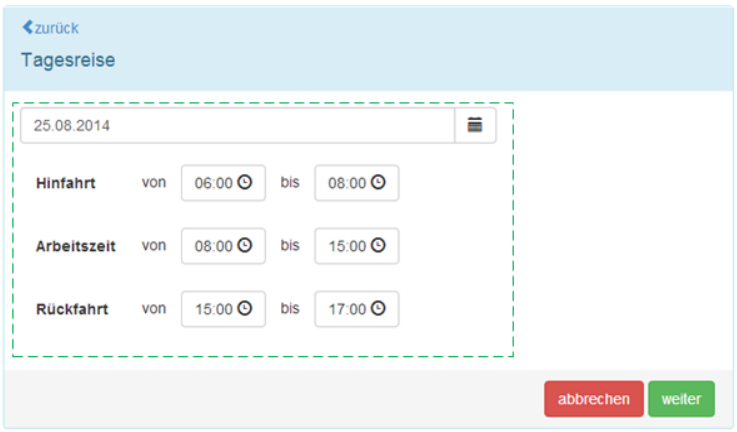

Abbildung 15: UI-Panel am Beispiel Tagesreise. Grün gestrichelter Bereich gehört nicht zur Direktive und wurde anschließend bei der Verwendung eingebunden.

Für die Vereinfachung, der Eingabe eines Datums, bietet AngularUI ein „Datepicker“-Element an. Um eine schnelle Wiederverwendung der Popup-Variante zu gewähren, kapselt die Direktive „popupDatepicker“ (Abbildung 16) genau diese. Jene gekapselte Datumseingabe kann mithilfe von Attributen von außen konfiguriert werden. Eine Standardkonfiguration ist innerhalb der Direktive innerhalb der „link“-Eigenschaft festgelegt.

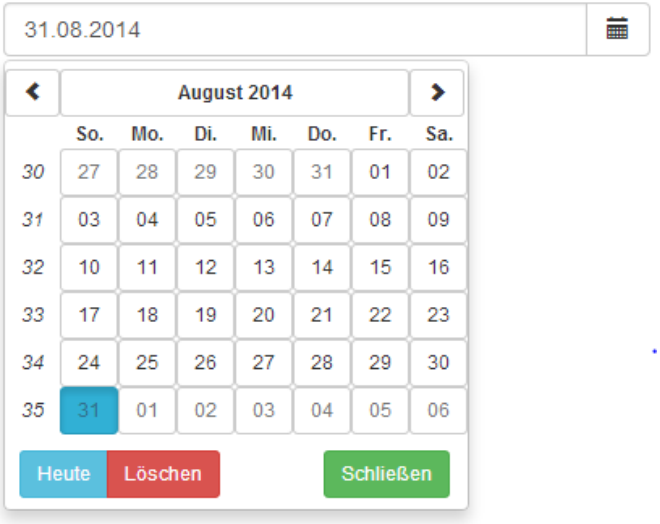

Abbildung 16: Popup-Datepicker

Im Kapitel 4.1 wurde ebenso die Wichtigkeit einer Zeiteingabe betont. Zu diesem Zweck implementiert die „popupTimepicker“ Direktive den AngularUI „Timepicker“. Da dieser jedoch gestaltungstechnisch nicht ganz in das Konzept passt, wird er innerhalb des neuen Widgets angepasst. Dazu wird die Zeiteingabe an einen Button durch ein „Popup“ angehängt. In Abbildung 17 ist dieses Element zu sehen.

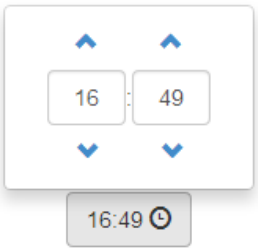

Abbildung 17: Popup-Timepicker

Ein weiterer technischer Kernaspekt ist die Eingabe von Zeiträumen. In diesem Zusammenhang existiert die Direktive mit den Namen „timeRange“. Sie fügt zwei der zuvor erläuterten „popupTimepicker“ zu einer Zeitraumeingabe zusammen. Das Besondere an ihr ist, dass sie eine eigene Validierung für die Zeitangaben besitzt. Dafür implementiert die Direktive einen eigenen Controller und verwendet den Hilfsservice „DateTimeService“.

Dieser Service bietet eine Reihe von Hilfsfunktionen für Datums- und Zeitangaben an. Unter anderem auch die besagte Validierung des Zeitraumes, durch Zuhilfenahme der Funktion .validadeTimeRange(*startTime*, *endTime*). Diese Funktion prüft, ob es sich bei beiden Übergabeparametern um Datumsangaben handelt und ob die Startzeit kleiner der Endzeit ist.

```
1  function isValidTimeRange(startTime, endTime) {
2    return angular.isDate(startTime)
3      && angular.isDate(endTime)
4      && startTime.getTime() <= endTime.getTime();
5  }
```

Listing 13: Beispiel Zeitvalidierung aus dem Service „DateTimeService“

Daneben bietet der Service auch noch die Funktion .adjustDateInTimeRange(*date*, *timeRange*) zur Anpassung des Datums einer Zeitspanne. Die Erstellung eines Nachfolgetages stellt durch die Funktion .createInitialisedDate(*previousDate*, *hourDifference*) auch kein Problem dar.

Dieses hier entwickelte Hilfsmodul erleichtert jetzt die Entwicklung des fachlichen Teils der Anwendung. Alle darin enthaltenen Elemente sind ohne Einschränkungen in anderen Modulen oder AngularJS-Anwendungen verwendbar.

4.2.4 travelTimeRegistration-Modul

Das Modul mit dem Namen „travelTimeRegistration“ bildet den fachlichen Kern der Reisezeitenerfassung. Sämtliche erreichbaren Oberflächen sind in diesem Modul implementiert. Auch das Routing wird über diesen Bestandteil definiert. So werden in der Datei travelTimeRegistration.js folgende Pfade festgelegt:

URI	Oberfläche
/	Startseite
/oneDayTravel	Tagesreise
/multiDayTravel	Mehrtagesreise
/evaluation	Auswertung

Tabelle 2: Routing der Applikation

Für die Darstellung und Verarbeitung der **Tagesreise** existieren der Controller „OneDayTravelCtrl“ und der Service „OneDayTravelService. Die Anbindung des Controllers erfolgt direkt im HTML-Template one-day-travel.html. Dieses Template verwendet nun die Direktiven aus Kapitel 4.2.3, um die Anforderungen[43] an die Eingabe der Reise- und Arbeitsdaten zu erfüllen. Der Controller (Listing 14, Seite 48) selbst besitzt einen überschaubaren Aufbau.

Die Logik, wie zum Beispiel Validierung und Upload, ist komplett im vom „OneDayTravelCtrl“ injizierten Service programmiert. Er verwendet unter anderem den im Utils-Modul entwickelten „DateTimeService“, sowie einen Backendservice[44]

Durch die Implementierung eines einzigen hauptverantwortlichen Service verringert sich die Anzahl der abhängigen Module im Controller. Erweiterungen und Änderungen sind leichter realisierbar. Ein Screenshot der Tagensreise ist in Abbildung 15 (Seite 44) zu finden.

[43] Siehe Kapitel 4.1.1
[44] Siehe Kapitel 4.2.5

```
1  angular.module('gisa.travelTimeRegistration')
2    .controller('OneDayTravelCtrl', ['$scope', '$location', 'OneDayTravelService',
3      function ($scope, $location, OneDayTravelService) {
4        OneDayTravelService.initOneDayTravel($scope);
5
6        $scope.save = function () {
7          OneDayTravelService.upload($scope)
8            .success(function (response, status) {
9              $location.path('/evaluation');
10           })
11           .error(function (response, status) {
12             $scope.error = "Fehler beim Speichern der Tagesreise. Status:" + status;
13           });
14       }
15
16       $scope.abort = function () {
17         $location.path('/');
18       };
19     }]);
```

Listing 14: OneDayTravelCtrl

Obwohl die **Mehrtagesreise** in ihren Anforderungen[45] wesentlich anspruchsvoller als die Tagesreise ist, können technisch viele Parallelen gezogen werden. Die Mehrtagesreise besteht ebenfalls hauptsächlich aus einer Template, einem Controller und einem dazugehörigen Service.

Neu hinzu kommt eine Direktive „DateRangeContainer“ zur Kapselung der Eingabe eines Arbeitstages für die Mehrtageserfassung. Folglich ist es einfacher, eine Liste von Arbeitstagen zu erzeugen.

Die Liste wird vom Controller beaufsichtigt. Sie befindet sich während der Eingabe, wie zu erwarten, komplett im Scope. Demzufolge erfolgt eine stetig aktuelle Ausgabe (bzw. Eingabemöglichkeit) in der View. Die Template gibt die vorher implementierte Direktive „DateRangeContainer“ über `ng-Repeat` aus. Listing 15 (Seite 49) zeigt, mit welch einfachen Mitteln eine komplexe Anzeige von Listenelementen mithilfe von AnuglarJS realisierbar ist. So ist in Listing 15 die Zeile 8 bis 11 für genau diese Ausgabe verantwortlich. Der Controller ist ebenfalls für sämtliche weiteren Aussteuerrungen zuständig.

[45] Siehe Kapitel 4.1.2

```
<div ng-controller="MultiDayTravelCtrl">
  <app-frame  title="Mehrtagesreise" error-msg="error" on-back="back()"
        on-abort="back()" on-continue="save()">
    <header>
      <date-range-container  title="Hinreise" date="outwardDate">
      </date-range-container>
    </header>
    <section  ng-repeat="workDay in workingDays | orderBy:'date'">
      <date-range-container  title="{{$index+1}}. Arbeitstag" date="workDay"
                  on-close="deleteWorkDay($index)"></date-range-container>
    </section>
    <button  style="..." class="btn btn-info btn-block"
        ng-click="addWorkDay()">Arbeitstag hinzufügen
    </button>
    <footer>
      <date-range-container  title="Rückreise" date="returnDate">
      </date-range-container>
    </footer>
  </app-frame>
</div>
```

Listing 15: Template der Mehrtagesreise (multi-day-travel.html)

Für die Ausführung der Logik ist wieder zum nicht unerheblichen Teil ein Service verantwortlich. Dieser „MutliDayTravelService“ liefert Funktionen zur Validierung, zum Upload und zur Erzeugung der Datenstruktur. Gleichermaßen injiziert er den Service, der den Client mit dem Backend koppelt.

Abschließend beinhaltet das Modul noch die Oberfläche für die „**Auswertung“** der Reisedaten. Diese tabellarische Ansicht besitzt einen überschaubaren Funktionsumfang. Dem zugrunde liegend wurde auf einen extra Service verzichtet und die wenige Anwendungslogik direkt im zuständigen Controller „EvaluationCtrl“ implementiert.

Ein Screenshot der Views für die Mehrtageserfassung (Abbildung 18) und der Auswertung (Abbildung 19) ist auf der Seite 50 zu finden. Schlussendlich sollte bedacht werden, dass das Modul „travelTimeRegistration“ nicht zur Weiterverwendung innerhalb weiterer Module konzipiert wurde. Mit Ausnahme des im nachfolgenden Kapitel (Seite 51) vorgestellten Service sind alle Elemente als private Bestandteile zu betrachten.

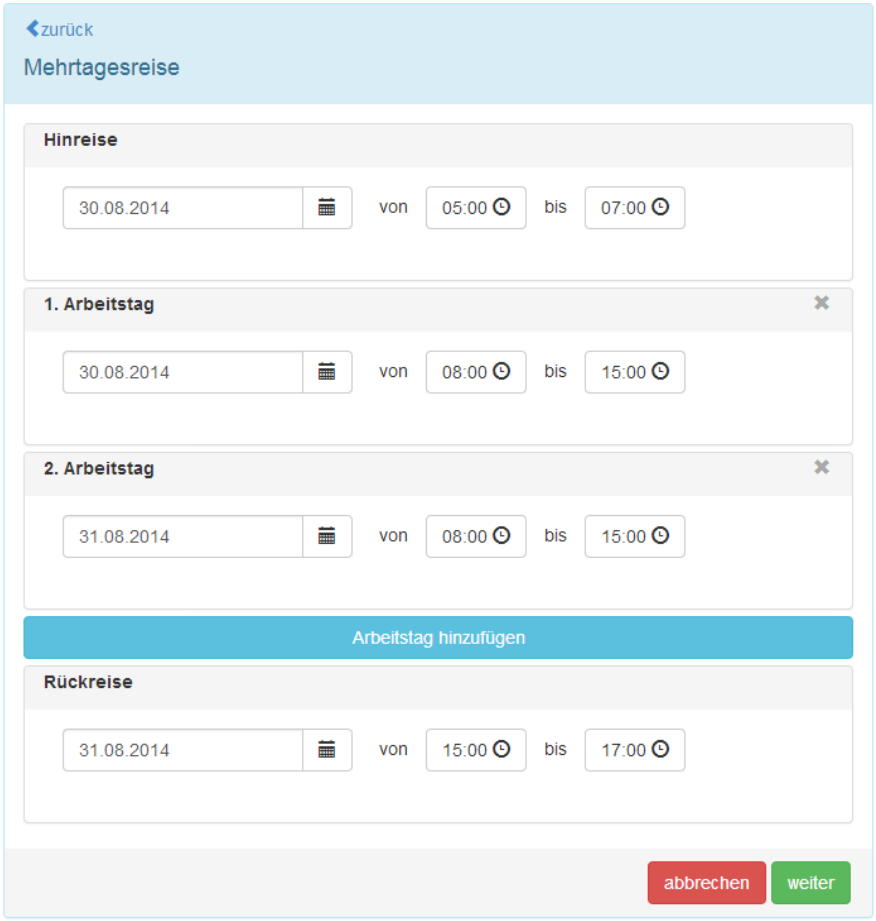

Abbildung 18: Oberfläche für die Erfassung der Mehrtagesreise

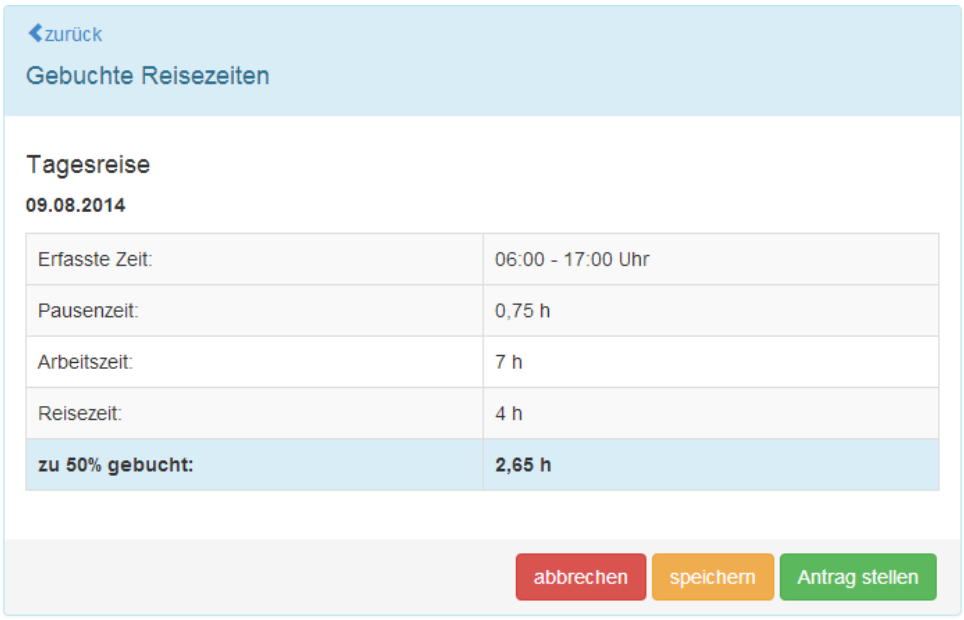

Abbildung 19: View der Auswertung

4.2.5 Backendanbindung

Ein zentraler, bisher weitestgehend vernachlässigter Teil der Applikation bzw. des Moduls „timeTravelRegistration“, ist die Anbindung an das Backend. Realisiert wird diese Verbindung über den Service „BackendService“, welcher die REST[46]-API implementiert.

Folglich muss zuerst die Schnittstelle betrachtetet werden, die das Backend anbietet (bzw. anbieten muss). Client und Server müssen, für eine funktionstüchtige Anwendung, zwingend in diesem Interface übereinstimmen.

URI	Methode	Nachricht	Antwort
/api/oneDayTravel	POST	JSON	-
/api/multiDayTravel	POST	JSON	-
/api/travelTime	GET	-	JSON
	POST	-	-
	PUT	-	-

Tabelle 3: REST-API

Die Tabelle 3 offenbart die existenten REST-Methoden. Die ersten beiden URIs werden verwendet, um die Daten der erfassten Reisezeiten im JSON-Format an das Backend zu senden. Der `GET`-Abruf der URI „/api/travelTime“ liefert die vom Server ausgewerteten Daten. Auffällig sind die `POST` und `PUT` Methoden dieses Abrufes, da sie keine relevanten Daten erhalten bzw. liefern. Ihr Zweck ist es, den Speichervorgang (`POST`) bzw. die Antragsstellung (`PUT`) in Gang zu setzen.

Der sich im Frontend befindende „BackendService“ implementiert diese API. Entsprechend bietet er eine Reihe von Funktionen (die ein Promise[47] zurückgeben) zur Realisierung der in Tabelle 3 aufgelisteten Abfragen an. Um dies zu bewerkstelligen, injiziert der Backend-Service den in Kapitel 3.7 beschrieben AngularJS-Service `$http`.

[46] Siehe Kapitel 2.6
[47] Siehe Kapitel 3.6

Eine Basis-URL kann mithilfe der Modulkonstante BACKEND_BASEPATH definiert werden. Aktuell befindet sich die Festlegung der besagten Konstante im Root-Modul der Webapplikation.

4.2.6 Tests

Um die Qualität der Anwendung zu wahren, wurde eine Reihe von Testfällen entwickelt. Allerdings beschränkt es sich hier auf Unit-Tests der Service-Implementationen. So wird zum Beispiel die Validierung der Tagesreise innerhalb eines Unit-Testfalls geprüft. Aufgrund der geringen Zahl der UI-Workflows und den in Kapitel 3.8 genannten Gründen wurde auf E2E-Test verzichtet.

Genauso wurde auf einen Test der Backendabfragen verzichtet. Die Korrektheit der REST-Schnittstelle sollte innerhalb der Backend-Entwicklung überprüft werden. Ein erneutes Testen innerhalb der Client-Anwendung wäre somit unnötiger zusätzlicher Aufwand. Das alleinige Testen der Richtigkeit der Backend-Servicemethoden würde eine Vielzahl von vorgetäuschten Abfragen und Daten benötigen. Der Aufwand steht hierfür kaum im Verhältnis zum Nutzen.

Bei allen sich innerhalb der Anwendung befindlichen Direktiven handelt es sich um Container. Aus diesem Grund werden sie nicht automatisch getestet. Solche Direktiven zu testen ist sehr aufwendig. Der komplette Kompiliervorgang müsste selbstständig angestoßen werden. Ein eigener Scope initialisiert und das Abrufen des Templates über templateUrl müsste vorgetäuscht werden. Nicht vergessen werden darf danach auch nicht die eigentliche Einrichtung der Testumgebung in Form eines Test-Templates. Der hierbei entstehende Aufwand ist immens. Eine manuelle Prüfung ist folgerichtig praktikabler. Obendrein wurden die für die Benutzereingabe relevanten AngularUI-Direktiven bereits von dessen Machern getestet.

Zur Realisierung der Tests finden die im Kapitel 4.2 (Seite 40) genannten Tools und Frameworks Anwendung. Getestet wird parallel auf den aktuellen Versionen der Browser Firefox, Internet Explorer und Chrome.

5 Fazit

Die Arbeit hat gezeigt, wie umfangreich und flexibel die Entwicklung von Single Page Webapplikationen ist. Die hohe Anzahl der Technologien scheint anfangs unüberschaubar und sogleich offenbart sich eine Vielzahl von Möglichkeiten. Folglich ist ein Verzicht auf einen breit gefächerten Einsatz von Frameworks kaum denkbar.

AngularJS erwies sich als äußerst nützlicher Helfer. Dank seiner umfangreichen API und der außergewöhnlichen Architektur wird die Realisierung von komplexen Webapplikationen stark vereinfacht. Leider folgt aus dessen Mächtigkeit ein nicht zu unterschätzender Einarbeitungsaufwand. Obwohl erste Erfolge schnell erzielt werden können, ist für die Erstellung umfangreicher Webanwendung ein nachhaltiges Verständnis über das Framework von Nöten. Wie diese Arbeit dessen ungeachtet zeigt, lohnt sich die Mühe. Besonders hervorzuhebende Stärken sind die Zwei-Wege-Datenbindung und die Option der Erstellung eigener HTML-Elemente durch Direktiven.

Gezeigt wurde darüber hinaus die Stärke der modularen Entwicklungsstruktur. Ein Ausbau, der in Rahmen dieser Bachelorarbeit entstanden Anwendung, ist dank ihres Modul geprägten Aufbaus kein Problem. Es ist demzufolge vorstellbar, die Webapplikation von einer kleinen Reisezeitenerfassung zu einem umfangreichen Mitarbeiterportal schrittweise auszubauen.

Das Potenzial dynamischer Single Page Webapplikationen ist außerordentlich hoch. Wie in Kapitel 4 gezeigt, ist die Darstellung von dynamischen benutzerspezifischen Inhalten eine besondere Stärke solcher Systeme. Der Unterschied zwischen einer echten Desktopanwendung zu einer Webapplikation in der Benutzerführung ist verschwindend gering. Es ist nicht undenkbar, dass solche SPAs Desktopprogramme vollständig ablösen.

Glossar

Asynchronous JavaScript and XML (Ajax):
Programmiermodell für die asynchrone Datenübermittlung.

AngularJS:
JavaScript-Framework zur Erstellung von SPA Webapplikationen.

Application Programming Interface (Api):
Schnittstelle zur Anwendungsentwicklung.

Backend:
Auf der Serverseite laufender Anwendungsbestandteil.

Blackbox:
System, das nur von außen sichtbar ist und keine detaillierten Erkenntnisse über seinen internen Aufbau liefert.

Bower:
Packetverwaltungstool für JavaScript Bibliotheken.

Cache:
Puffer-Speichers eines Browser.

Callback:
Funktion, die als Parameter an eine andere Funktion übergeben wurde.

Cascading Style Sheets (CSS):
Sprache zur Formatierung von Dokumenten.

Dependency Injection (DI):
Entwurfsmuster der Softwareentwicklung zur Verringerung von Abhängigkeiten der Programmbestandteile.

Testgetriebene Entwicklung (TDD):
Vorgehensweise bei der Softwareentwicklung, bei der die Tests zuerst implementiert werden.

Document Object Model (DOM):
Schnittstelle für den Zugriff auf XML- und HTML-Dokumente.

Flash:
Eine von Adobe entwickelte Plattform zur Präsentation von multimedialen und interaktiven Inhalten.

Frontend:
Auf der Benutzerseite laufender Anwendungsbestandteil.

Graphical User Interface (GUI):
Grafische Benutzeroberfläche. Schnittstelle für Interaktionen des Benutzers mit dem Computer.

Hypertext Markup Language (HTML):
Auszeichnungssprache für (Web-)Dokumente.

Intranet:
Nicht öffentliches Rechnernetzwerk für innerbetriebliche Zwecke.

Java-Applet:
Im Webbrowser über ein Plug-in direkt ausgeführtes Java-Programm.

JavaScript:
Prototypenbasierte Skriptsprache zur Entwicklung von Webapplikationen.

JavaScript Object Notation (JSON):
Ein kompaktes objektorientiertes Datenaustauschformat.

JavaServer Faces (JSF):
Java Spezifikation zur Entwicklung von Webanwendung auf Basis von Servlets und JavaServer Pages.

JavaServer Pages (JSP):
Web-Programmiersprache für die Entwicklung von dynamischen Benutzeroberflächen eines Webservers.

JQuery:
JavaScript-Bibliothek für DOM-Manipulationen.

Objektorientierte Programmierung (OOP):
Programmierparadigma, das ein System durch Objekte beschreibt.

Plug-in:
Softwaremodul, das eine Anwendung um weitere Funktionalitäten erweitert.

Silverlight:
Erweiterung des Browsers für die Ausführung von dynamischen Webapplikationen.

Singleton:
Entwurfsmuster der Softwareentwicklung. Genau gesagt handelt es sich um ein Erzeugungsmuster das sicherstellt, dass nur eine einzige Instanz eines Objektes vorhanden ist.

Stylesheet:
Datei, die Formatierungsvorschriften für Dokumente enthält.

Routing:
Zuordnung der Webpfade zu den entsprechenden Oberflächen.

Template:
Vorlage (Schablone), die mit Inhalten gefüllt werden kann.

UI-Widget:
Steuerelement für grafische Benutzeroberflächen.

Uniform Resource Identifier (URI):
Identifikator für Ressourcen von Websystemen.

Uniform Resource Locator (URL):
Lokalisator für Ressourcen von Websystemen.

Whitebox:
Softwaresystem, dessen interne Funktionen bekannt sind.

XMLHttpRequest:
Verfahren für die Übermittlung von Daten über HTTP.

Quellenverzeichnis

[AngularJS-DI] AngularJS Developer Guide – Dependency Injection: *https://docs.angularjs.org/guide/di* (Aufgerufen am 16.06.2014)

[AngularJS-E2E] AngularJS Developer Guide – E2E Tests: https://code.angularjs.org/1.2.18/docs/guide/e2e-testing (Aufgerufen am 28.07.2014)

[AngularJS.de-DB] AngularJS.de Buch – Databinding: http://angularjs.de/buch/databinding (Aufgerufen am 08.07.2014)

[AngluarJS.de-EG] AngularJS.de Buch – Was ist AngularJS?: *http://angularjs.de/buch/was-ist-angularjs* (Aufgerufen am 07.07.2014)

[Barth 2012] Michael Barth: *Prinzipien der Modularisierung*. Universität Ulm 2012

[Bayer 2002] Thomas Bayer: *REST Web Services*. Orientation in Objects 2012

[Burnham 2012] Trevor Burnham: Async JavaScript. The Pragmatic Programmers 2012

[Developa] Developa Blog – Frontend-Entwicklung webbasiert oder nativ?: *http://www.developa.org/Frankfurt/html5-javascript-oder-doch-lieber-nativ/* (Aufgerufen am 17.06.2014)

[Franke & Ippen 2012] Florian Franke, Johannes Ippen: *Apps mit HTML5 und CSS3*. Galileo Press. 1. Auflage 2012

[Goll und Dausmann 2013] Joachim Goll, Manfred Dausmann: *Architektur- und Entwurfsmuster der Softwaretechnik*. Springer Vieweg. 2013

[Green & Seshadri] Brad Green und Shyam Seshadri: *AngularJS*. O'Reilly 2013

[Horn] Thorsten Horn – Architektur von Webanwendugen: *http://www.torsten-horn.de/techdocs/webanwendungen.htm* (Aufgerufen am 3.07.2014)

[Isernhagen und Helmke 2004] Rolf Isernhagen, Hartmut Helmke: *Softwaretechnik in C und C++*. Carl Hanser Verlag. 4. Auflage 2004

[JavaScript heute] iX Developer: *JavaScript heute*. Heise Zeitschriften Verlag GmbH & Co. KG. 1/2014

[JavaScript.Info] JavaScript Tutorial – Events and timing in-deph: http://javascript.info/tutorial/events-and-timing-depth#javascript-execution-and-rendering (Aufgerufen am 22.07.2014)

[Long Le] Long Le‘s Blog – Modern Web Application Layered High Level Architecture with SPA, MVC, Web API, EF, Kendo UI, OData : http://blog.longle.net/2013/06/13/modern-web-application-layered-high-level-architecture-with-spa-mvc-web-api-ef/ (Aufgerufen am 03.07.2014)

[Müller 2010] Bernd Müller: JavaServer Faces 2.0 . Carl Hanser Verlag. 2010

[Neumann & Murmann 2001] Heiko Neumann, Klaus Murmann: *Script Allgemeine Informatik II – Teil V: Software-Module*. Universität Ulm 2001

[Osmani] Addy Osmani - Building Single Page Applications With jQuery’s Best Friends: *http://addyosmani.com/blog/building-spas-jquerys-best-friends/* (Aufgerufen am 01.07.2014)

[Osten 2009] Antje Osten: *Identifikation von Komponenten*. 2009

[PDS Blog] Prinzipien der Softwaretechnik – Das Modularisierungs-prinzip: *http://prinzipien-der-softwaretechnik.blogspot.de/2013/11/das-modularisierungsprinzip.html* (Aufgerufen am 18.06.2014)

[Rainer Oechsle 2013] Rainer Oechsle: *Java – Komponenten*. Carl Hanser Verlag. 2013

[SPA Book] Single page apps in depth – goal: *http://singlepageappbook.com/goal.html* (Aufgerufen am 19.06.2014)

[Spindler 2014] Heiko Spindler: *Single-Page-Web-Apps*. Franzis Verlag. 2014

[Swenson-Healey 2013] Erin Swenson-Healey - The JavaScript Event Loop: Explained: *http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/* (Aufgerufen am 22.07.2014)

[Tarasiewcz & Böhm] Philipp Tarasiewicz, Robin Böhm: *AngularJS – Eine praktische Einführung in das JavaScript-Framework*. dpunkt.verlag 2014

[Theis 2014] Thomas Theis: *Einstieg in JavaScript*. Galileo Press. 1. Auflage 2014

[Ulf Fildebrandt 2012] Ulf Fildebrandt: *Software modular bauen*. dpunkt.verlag 2012

[Vermeersch 2014] Ruben Vermeersch – Testing with Anuglar.JS: http://savanne.be/articles/testing-with-angular-js/ (Aufgerufen am 22.07.2014)

[Vishia] Vishia – Modulare Programmierung und Abhängigkeiten: *http://www.vishia.org/SwEng/html/Dependencies.html* (Aufgerufen am 16.06.2014)

[Wikipedia 1] Wikipedia – HTML5: *http://en.wikipedia.org/wiki/HTML5* (Aufgerufen am 18.03.2014)

[Wikipedia 2] Wikipedia – JavaScript: *http://en.wikipedia.org/wiki/JavaScript* (Aufgerufen am 18.03.2014)

[Wikipedia 3] Wikipedia – Prototypenbasierte Programmierung: *http://de.wikipedia.org/wiki/Prototypenbasierte_Programmierung* (Aufgerufen am 01.06.2014)

[Wikipedia 4] Wikipedia – Model View ViewModel: *http://en.wikipedia.org/wiki/Model_View_ViewModel* (Aufgerufen am 01.06.2014)

[Wikipedia 5] Wikipedia – Model View ViewModel: *http://de.wikipedia.org/wiki/Model_View_ViewModel* (Aufgerufen am 01.06.2014)

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Inhalt der CD

- Bachelorarbeit als PDF
- Quellcode der Reisezeitenerfassung
- Ausführbare Version der Reisezeitenerfassung (mit simulierten Backend)